UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

FRANCO BRUNETTA PAN

DESENVOLVIMENTO DE EXTENSÃO DE VS CODE PARA A ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

CAXIAS DO SUL

2024

FRANCO BRUNETTA PAN

DESENVOLVIMENTO DE EXTENSÃO DE VS CODE PARA A ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Ricardo Vargas Dorneles

CAXIAS DO SUL

2024

FRANCO BRUNETTA PAN

DESENVOLVIMENTO DE EXTENSÃO DE VS CODE PARA A ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado(a) em 06/12/2024

BANCA EXAMINADORA

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

Prof. Me. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul - UCS

Prof. Dr. Leonardo Dagnino Chiwiacowsky
Universidade de Caxias do Sul - UCS

AGRADECIMENTOS

Agradeço aos meus pais, Gilson e Isabela, e aos meus familiares pelo apoio incondicional, pelo amor e incentivo em todos os momentos.

Agradeço também ao meu orientador pelo conhecimento compartilhado, pela paciência e dedicação que foram fundamentais para o desenvolvimento desta monografia.

Aos meus amigos e colegas de classe, agradeço pela ajuda mútua, pelos momentos de descontração e pelas experiências compartilhadas.

A todos que, direta ou indiretamente, fizeram parte da minha jornada acadêmica, o meu muito obrigado.

RESUMO

Compreender como os algoritmos se comportam em termos de tempo de execução é essencial para otimizar o desempenho de software. No entanto, a análise teórica da complexidade não é fácil nem diretamente aplicada na prática, pois os desenvolvedores enfrentam desafios ao estimar a eficiência de seus códigos sem ferramentas adequadas. Para suprir essa necessidade, foi implementada uma extensão para o Visual Studio Code que analisa a complexidade de algoritmos, integrando-se diretamente ao processo de desenvolvimento de software.

Este trabalho concentrou-se em desenvolver uma solução prática, após uma revisão teórica e avaliação de soluções existentes. Os resultados obtidos mostraram que a extensão consegue analisar estruturas básicas de controle de fluxo ao utilizar Abstract Syntax Tree (AST) e resultados mais assertivos por modelos de linguagem natural, fornecendo estimativas assintóticas corretas. A análise automatizada demonstrou ser eficaz, preenchendo a lacuna entre a teoria e a prática, e permitindo que desenvolvedores e alunos avaliem rapidamente a complexidade de seus algoritmos, chegando a cerca de 96% de acerto.

A validação da solução desenvolvida confirmou que a análise integrada ao fluxo de desenvolvimento é não apenas viável, mas também benéfica, promovendo uma maior conscientização sobre a complexidade computacional e incentivando boas práticas de codificação, facilitando a aplicação prática dos conceitos teóricos de complexidade.

Palavras-chave: plugin; complexidade; algoritmo; extensão; big o; assintótico.

ABSTRACT

Understanding how algorithms behave in terms of execution time is essential for optimizing software performance. However, the theoretical analysis of complexity is not easy or directly applicable in practice, as developers face challenges in estimating the efficiency of their code without adequate tools. To address this need, an extension for Visual Studio Code was implemented that analyzes the complexity of algorithms, integrating directly into the software development process.

This work focused on developing a practical solution after a theoretical review and evaluation of existing solutions. The results obtained showed that the extension can analyze basic control flow structures using AST and provide more accurate results through natural language models, delivering correct asymptotic estimates. The automated analysis turned out to be effective, bridging the gap between theory and practice, and allowing developers and students to quickly assess the complexity of their algorithms, achieving around 96% accuracy.

The validation of the developed solution confirmed that the integrated analysis into the development workflow is not only feasible but also beneficial, promoting greater awareness of computational complexity and encouraging good coding practices, thus facilitating the practical application of theoretical concepts of complexity.

Keywords: plugin; complexity; algorithm; extension; big o; asymptotic.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

ANAC Analisador de Complexidade

API Application Programming Interface

AST Abstract Syntax Tree

CC Coeficiente de Classificação

COStA COSt and Termination Analyzer for Java Bytecode

CPU Central Processing Unit

FC Fator de Classificação

GPU Graphics Processing Unit

HTML Hypertext Markup Language

IA Inteligência Artificial

ICNTE International Conference on Nascent Technologies in Engineering

IDE Integrated Development Environment

IGED Interpretador Gráfico de Estrutura de Dados

JSON JavaScript Object Notation

LSP Language Server Protocol

LLM Large Language Model

MaLiJan Maximum Likelihood Java Bytecode Analyzer

MOCCA Measurement of Complexity of an Certain Algorithm

NPM Node Package Manager

NPU Neural Processing Unit

SDK Software Development Kit

URL Uniform Resource Locator

UNIVALI Universidade do Vale do Itajaí

LISTA DE SÍMBOLOS

O	*Big* O
Ω	*Big* Ômega
θ	*Big* Teta
Φ	Função Potencial

SUMÁRIO

1 INTRODUÇÃO

Os algoritmos são essenciais na construção de soluções de *software* e, embora a capacidade de processamento continue crescendo nos últimos 60 anos (INTEL, 2024), o consumo eficiente dos recursos é essencial para a redução de custos financeiros e da emissão de carbono. Considerando essa problemática, a análise da complexidade computacional de algoritmos é vital.

Não foram encontradas ferramentas de fácil acesso que analisam a complexidade computacional de um algoritmo diretamente no ambiente de desenvolvimento. Quando um desenvolvedor deseja obter a complexidade assintótica, o processo é, em geral, realizado manualmente. Essa lacuna motiva este trabalho, que propõe a criação de uma extensão para o editor VS Code[1] , automatizando essa análise. A ferramenta oferecerá uma estimativa da complexidade assintótica de algoritmos, facilitando a verificação durante o desenvolvimento, promovendo códigos mais eficientes e contribuindo para o aprimoramento do fluxo de trabalho.

1.1 OBJETIVOS

O objetivo principal deste trabalho é implementar uma extensão para o VS Code capaz de avaliar, de forma estática, a complexidade assintótica de algoritmos escritos em linguagem de programação C. Para isso, propõe-se:

- Desenvolver algoritmos que analisem estruturas de controle de fluxo (laços, condicionais, chamadas recursivas) para estimar a complexidade assintótica, por exemplo $O(1)$, $O(n)$, $O(n^2)$;
- Integrar a análise ao ambiente do VS Code, oferecendo feedback direto ao desenvolvedor;
- Incorporar ferramentas auxiliares baseadas em *machine learning*, como LLMs, para complementar a análise estática.

1.2 JUSTIFICATIVA

A popularidade do VS Code, aliada à ausência de ferramentas que realizam a análise de complexidade assintótica integradamente ao desenvolvimento, torna o tema deste trabalho relevante.

Embora existam perfiladores e ferramentas de análise de desempenho, esses métodos são, em geral, utilizados sobre o código compilado ou em execução, sendo aplicados após o

[1] MICROSOFT. **Your First Extension | Visual Studio Code Extension API**. Microsoft, 2024. Disponível em: <https://code.visualstudio.com/api/get-started/your-first-extension>. Acesso em: 27 fev. 2024.

desenvolvimento. Este trabalho visa preencher essa lacuna, trazendo a análise para o momento em que o código é escrito, com impacto direto na qualidade e eficiência do desenvolvimento de software.

1.3 HIPÓTESE

A hipótese deste trabalho é que a análise automatizada da complexidade computacional pode substituir a avaliação manual, oferecendo resultados precisos e integrados ao fluxo de desenvolvimento. Além disso, espera-se que essa análise promova maior conscientização dos desenvolvedores sobre os impactos de desempenho do código.

1.4 METODOLOGIA

A metodologia adotada segue as etapas abaixo, que se interligam para atingir os objetivos propostos:

1.4.1 Revisão Bibliográfica e Teórica

- Levantamento de conceitos sobre análise de complexidade computacional, incluindo revisão de livros didáticos e artigos científicos;
- Estudo de guias oficiais para o desenvolvimento de extensões do VS Code;
- Revisão de trabalhos relacionados, analisando abordagens e lacunas existentes.

1.4.2 Levantamento de Requisitos

- Identificação de funcionalidades essenciais por meio de entrevistas com desenvolvedores;
- Definição do escopo funcional da extensão, delimitando linguagens suportadas e grau de detalhamento das análises.

1.4.3 Prototipação

- Desenvolvimento de uma versão simplificada da extensão para validar conceitos técnicos;
- Implementação inicial de algoritmos básicos de análise de complexidade.

1.4.4 Avaliação e Validação

- Comparação dos resultados da extensão com análises realizadas manualmente por especialistas.

1.5 RESULTADOS ESPERADOS

Espera-se que este trabalho resulte em uma extensão funcional para o VS Code, capaz de: analisar estruturas básicas de controle de fluxo e fornecer estimativas assintóticas corretas; integrar-se ao ambiente do desenvolvedor intuitivamente, com impacto positivo no fluxo de trabalho e promover maior conscientização sobre complexidade computacional e boas práticas de codificação.

1.6 ESTRUTURA

Este trabalho está organizado da seguinte maneira: o Capítulo 2, “Revisão teórica”, apresenta o embasamento teórico relacionado ao tema deste projeto, incluindo conceitos de complexidade de algoritmos. O Capítulo 3, “Estado da arte”, discute ferramentas similares e suas limitações. O Capítulo 4, “Desenvolvimento”, detalha o planejamento e o processo realizado para a criação da extensão, incluindo o *design*, metodologias e validação. O Capítulo 5, "Testes e Validação da Extensão", avalia os resultados encontrados e analisa a validação perante a hipótese e os objetivos do projeto. Por fim, o Capítulo 6, “Considerações finais”, resume os principais pontos do trabalho e discute suas limitações e possíveis ações futuras.

2 REVISÃO TEÓRICA

Para providenciar a análise de complexidade de um algoritmo, precisa-se formalizar primeiramente a definição de um algoritmo e, posteriormente, como se deseja atingir tal resultado, passando igualmente pela definição de complexidade.

2.1 ALGORITMOS

Segundo Sherine *et al.* (2023, pg. 1), "um algoritmo é um conjunto finito de instruções que realiza uma tarefa específica". Já Knuth (1997, pg. 4), no seu livro *The Art of Computer Programming*, expande essa afirmação, iniciando o seu argumento declarando que algoritmos vão além, possuindo cinco características principais particulares:

a) finitude: um algoritmo precisa sempre terminar após um número finito de passos;

b) definição: determina que cada passo de um algoritmo seja definido precisamente, sem ambiguidade. Eis o motivo de algoritmos serem comumente expressos em linguagens de programação formalmente definidas;

c) entradas: são o conjunto de dados fornecidos ao algoritmo antes ou durante a execução;

d) saídas: são o conjunto de dados providenciados pelo algoritmo até o fim da execução, que possuem relação com as entradas;

e) eficácia: todas as operações de um algoritmo devem ser básicas o suficiente para serem executadas exatamente e em um tempo finito.

Esses princípios formam o sustentáculo do *design* e análise de algoritmos, guiando o desenvolvimento de *softwares* eficientes e confiáveis.

2.2 COMPLEXIDADE DE ALGORITMOS

É possível separar a complexidade em algumas categorias: a complexidade de código, que se refere ao código ser complicado de desenvolver prestar manutenção ou testar; a de espaço, relativa ao crescimento do uso de espaço em disco e memória; a complexidade computacional no tempo, sendo o foco deste trabalho.

2.2.1 Complexidade de código

A complexidade de código se refere à complexidade cognitiva para a compreensão por um humano. Uma das análises mais populares é a Complexidade Ciclomática descrita por McCabe (1976). Esse método tem o foco em testabilidade, leitura e manutenção de um código,

não necessariamente se atendo à complexidade computacional. Esse método consiste em isolar segmentos de um código em um dígrafo, no qual o número ciclomático $v(G)$ de um grafo G com n vértices, e arestas e p componentes conectados é $v(G) = e - n + p$.

2.2.2 Complexidade de espaço

A complexidade de espaço refere-se à quantidade de memória adicional que um algoritmo requer em relação ao tamanho da entrada. Essa medida avalia a eficiência do algoritmo em termos de uso de memória.

2.2.3 Complexidade computacional de tempo

A complexidade computacional pode ser medida perante diversos fatores, como tempo e espaço (SHERINE *et al.*, 2023). Como exemplo, pode-se relacionar isso ao espaço de alocação em memória e à velocidade de acesso aos dados (TOSCANI; VELOSO, 2012, p.15). Mas o fator cujo trabalho tem objetivo de analisar é na complexidade de tempo.

O tempo para a execução de um algoritmo pode ser medido pelo número de execuções de um determinado conjunto finito de operações. A esse conjunto é dado o nome de operação fundamental, considerando que a quantidade de execuções da operação fundamental serve de representação da quantidade potencial de trabalho de um determinado algoritmo.

Esse valor pode ser medido em relação ao tamanho da entrada. Regularmente, à medida que o tamanho da entrada aumenta, a quantidade de operações também tende a aumentar. No entanto, interpretar o tamanho da entrada pode ser complicado, pois ele pode ter significados diferentes, dependendo do contexto. Para um valor de entrada específico, o tamanho pode se referir (mas não se limitando) a:

a) o valor do comprimento de uma lista;

b) o tamanho da ordem de uma matriz;

c) o número de vértices e arestas de um grafo (CORMEN *et al.*, 2022, p.57);

d) uma operação matemática, em que, se a entrada for um valor escalar, o tamanho da entrada será o número de bits necessários para representá-lo (CORMEN *et al.*, 2022, p.57). Por exemplo, o número 8, representado como 1000 em binário, tem um tamanho de entrada de 4, por precisar de 4 bits. No entanto, em certos contextos, como ao calcular o fatorial, o tamanho da entrada é o próprio número.

2.2.3.1 Complexidade média

A complexidade média de um algoritmo pode ser definida como a média ponderada da eficiência de cada entrada, considerando a probabilidade da sua ocorrência (TOSCANI; VELOSO,

2012, p.100), obtendo-se, assim, a equação

$$C_{avg}(n) = \sum_{i=1}^{N} p_i \cdot C_i(n) \tag{2.1}$$

em que $C_{avg}(n)$ representa a complexidade média para uma entrada de tamanho n. N é o número total de entradas, p_i é a probabilidade da i-ésima entrada ocorrer e $C_i(n)$ é a complexidade para a (*i*)-ésima entrada.

2.2.3.2 Complexidade pessimista

A complexidade pessimista é calculada com base no pior cenário de execuções pela entrada (TOSCANI; VELOSO, 2012, p.42). As equações

$$\overset{=}{C}_p = \max_{d\,:\,\text{tam}(d)=n} \left(\frac{\text{desemp}(a,d)}{\text{tam}(d)} \right) \tag{2.2}$$

e

$$\overset{\leq}{C}_p = \max_{d\,:\,\text{tam}(d)\leq n} \left(\frac{\text{desemp}(a,d)}{\text{tam}(d)} \right) \tag{2.3}$$

providenciadas por Toscani e Veloso (2012, p.42) demonstram esse cálculo em que C_p representa a complexidade pessimista, d é a entrada do algoritmo, $tam(d)$ indica o tamanho da entrada e desemp(a, d) é o custo da execução do algoritmo "a" para uma entrada "d" (equivalente a custo(exec(a, d))).

Os motivos pelos quais esse método é frequentemente utilizado, e pelo qual o trabalho foca na complexidade pessimista (na sua versão assintótica), são fornecidos por Cormen *et al.* (2022, p.61) e Sherine *et al.* (2023, p.16):

a) o limite superior fornece segurança de que o algoritmo nunca será pior do que o caso encontrado;

b) o pior caso pode ocorrer frequentemente, como nos casos em que apenas dados não presentes em cache são referenciados e uma consulta em banco é necessária;

c) o caso médio pode ser o pior caso (por exemplo: busca linear em lista não ordenada);

d) alguns sistemas exigem que o usuário forneça um limite superior para a quantidade de tempo que o programa irá rodar. Uma vez que este limite superior é atingido, o programa é abortado;

e) o programa pode precisar fornecer uma resposta satisfatória em tempo real;

f) se existem maneiras alternativas de resolver um problema, então a decisão sobre qual usar será baseada principalmente na diferença de desempenho esperada entre essas soluções.

2.2.3.3 Complexidade assintótica

O cálculo de complexidade assintótica é uma abordagem comum para analisar a complexidade de um algoritmo. Ela descreve o crescimento da complexidade em função do tamanho da entrada, utilizando uma cota assintótica, sendo mais prática do que as fórmulas e expressões anteriormente vistas.

Uma cota assintótica superior é uma função que cresce mais rapidamente do que outra (TOSCANI; VELOSO, 2012). Em um gráfico no qual o eixo horizontal representa o tamanho da entrada e o eixo vertical a complexidade, uma função assintótica superior ultrapassará outra no eixo vertical em algum ponto e não será mais ultrapassada, indicando um maior crescimento. Uma cota assintótica superior pode ser definida conforme a expressão

$$(\exists n_0 \in \mathbb{N})\,(\forall n \geq n_0) : f(n) \leq g(n) \tag{2.4}$$

em que existe algum número natural n_0 tal que para todos os n maiores ou iguais a n_0, a função $f(n)$ é sempre menor ou igual à função $g(n)$. Algumas das notações utilizadas para representar cotas assintóticas são:

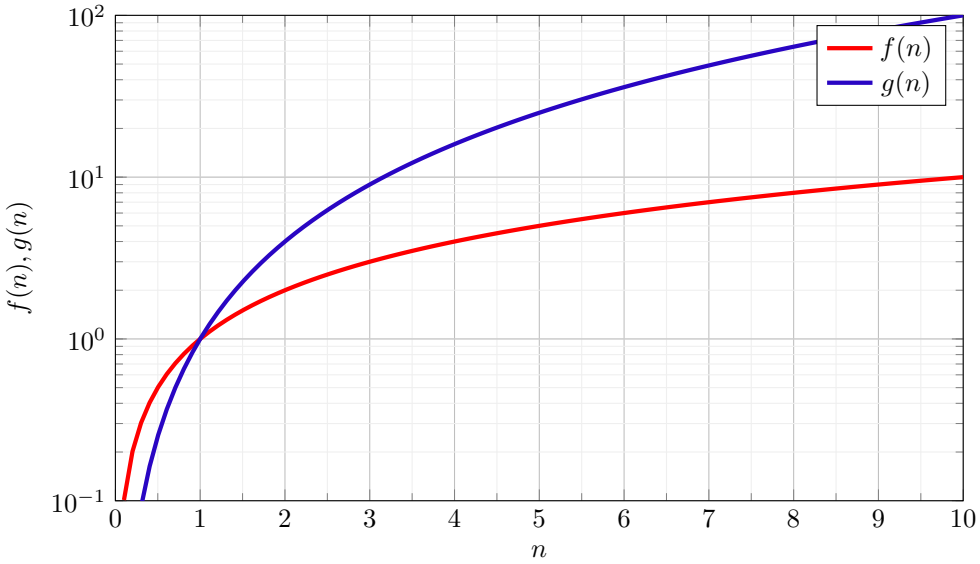
a) a notação ***Big* O (O)** define uma "cota assintótica superior a menos de constantes" (TOSCANI; VELOSO, 2012). Ela é usada para descrever o comportamento de uma função quando o tamanho da entrada se aproxima do infinito. Ao se dizer que uma função $f(n)$ é $O(g(n))$, isso significa que existe uma constante positiva c e um valor n_0 tal que $0 \leq f(n) \leq c \cdot g(n)$ para todos $n \geq n_0$. Em outras palavras, $f(n)$ é limitada superiormente por $g(n)$ multiplicada por uma constante, para n suficientemente grande. Esse comportamento é descrito pela expressão

$$(\exists c \in \mathbb{R}_+)\,(\exists n_0 \in \mathbb{N})\,(\forall n \geq n_0) : f(n) \leq c \times g(n) \tag{2.5}$$

e pela Figura 1. Assim, pode-se comparar a eficiência de diferentes algoritmos ignorando constantes e termos de menor ordem. A expressão (2.5) mostra que existe algum número real positivo (c) e um número natural n_0 tal que para todos os n maiores ou iguais a n_0, a função $f(n)$ é sempre menor ou igual a c vezes a função $g(n)$;

b) A notação ***Big* Ômega (Ω)** define uma "cota assintótica inferior a menos de constantes" (TOSCANI; VELOSO, 2012). Em outras palavras, existe um número real positivo d e um número natural n_0 tal que, para todo n maior ou igual a n_0, o valor da função $g(n)$ é maior ou igual ao valor da função $f(n)$ multiplicado por d. Essa é a definição formal de $g(n) = \Omega(f(n))$ e significa que $g(n)$ não cresce mais devagar do que $f(n)$ quando n se aproxima do infinito. Em outras palavras, não importa quão grande n se torne, $g(n)$ nunca será menor do que $d \times f(n)$ para algum valor constante d, assumindo que d é um número real positivo. Isso significa que $g(n)$ tem pelo menos

Figura 1 – Gráfico Big O comparando funções $f(n)$ e $g(n)$.



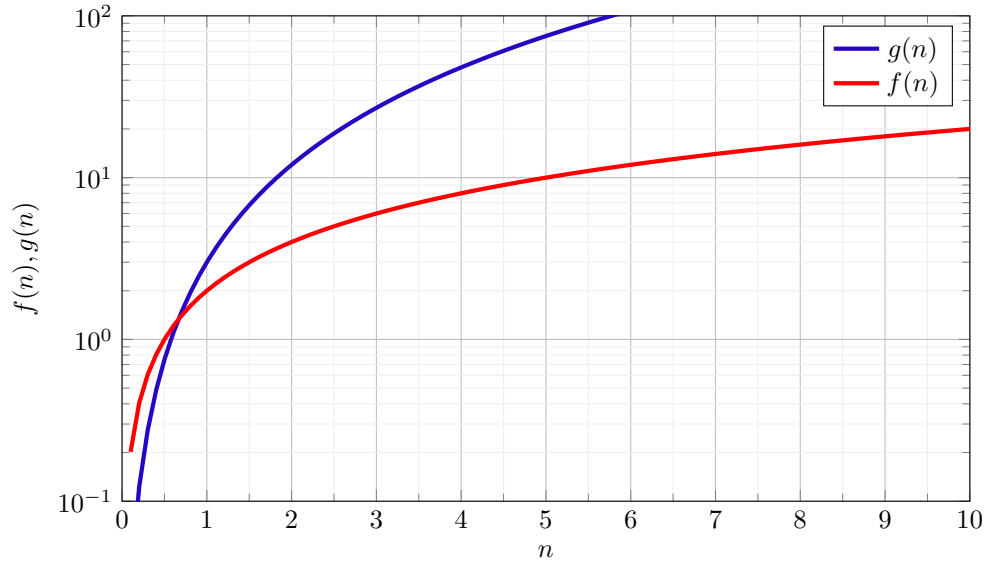
Fonte: O Autor (2024)

o mesmo "ritmo de crescimento" que $f(n)$ quando n se aproxima do infinito. Isso é descrito conforme a expressão:

$$(\exists d \in \mathbb{R}_+)\,(\exists n_0 \in \mathbb{N})\,(\forall n \geq n_0) : g(n) \geq d \times f(n) \tag{2.6}$$

e a Figura 2. A expressão (2.6) mostra que existe algum número real positivo d e um número natural n_0 tal que, para todos os n maiores ou iguais a n_0, a função $g(n)$ é sempre maior ou igual a d vezes a função $f(n)$.

Figura 2 – Gráfico Big Ômega comparando funções $f(n)$ e $g(n)$.



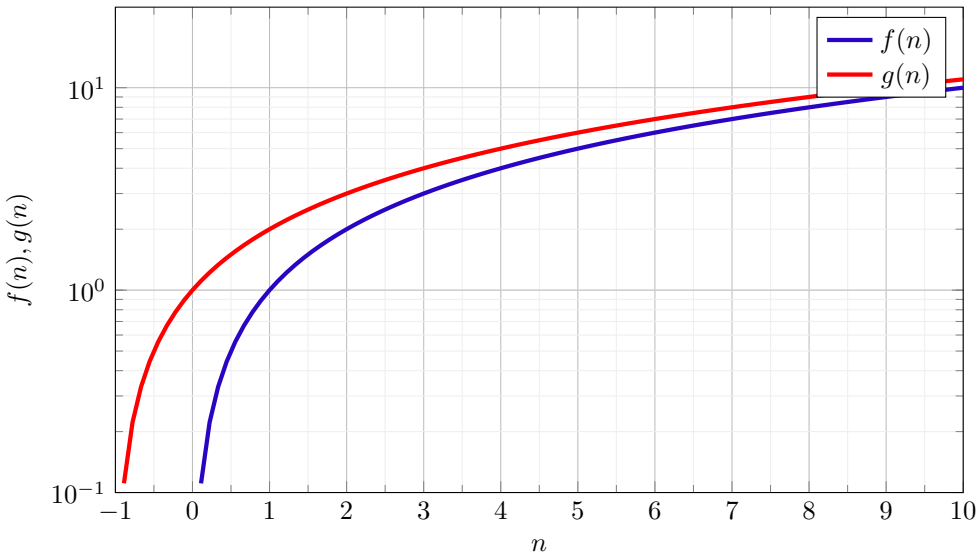
Fonte: O Autor (2024)

c) a notação ***Big* Teta** (Θ) define uma "cota assintótica apertada" (TOSCANI; VELOSO, 2012). Isso significa que uma função $g(n)$ é $\Theta(f(n))$ se existem constantes positivas $c1$, $c2$, e n_0, tais que $c1 \cdot f(n) \leq g(n) \leq c2 \cdot f(n)$ para todos $n \geq n_0$. Em outras palavras, $g(n)$ é limitada tanto superiormente quanto inferiormente por $f(n)$ multiplicada por uma constante, para n suficientemente grande. Isso significa que $g(n)$ tem o mesmo ritmo de crescimento que $f(n)$ quando n se aproxima do infinito. Isso é descrito conforme a expressão

$$(\exists c1, c2 \in \mathbb{R}_+)\,(\exists n_0 \in \mathbb{N})\,(\forall n \geq n_0) : c1 \times f(n) \leq g(n) \leq c2 \times f(n) \qquad (2.7)$$

e a Figura 3. A expressão (2.7) mostra que existem dois números reais positivos $c1$ e $c2$ e um número natural n_0 tal que para todos os n maiores ou iguais a n_0, a função $g(n)$ é sempre maior ou igual a $c1$ vezes a função $f(n)$ e sempre menor ou igual a $c2$ vezes a função $f(n)$.

Figura 3 – Gráfico Big Teta comparando funções $f(n)$ e $g(n)$.



Fonte: O Autor (2024)

2.2.3.4 Complexidade amortizada

Segundo Tarjan (1985, p.1), a avaliação amortizada da complexidade pode ser definida como uma complexidade avaliada pelo tempo sobre uma sequência de operações, ou seja, calculando a média dos tempos de execução das operações em uma sequência ao longo dela.

Existem algoritmos que, pela característica do seu funcionamento e da estrutura de dados utilizada, uma operação ou apenas algumas operações dentre N possuem complexidade demasiadamente superior. Isso implica que a avaliação pessimista sobre a operação de maneira individual pode resultar em uma estimativa excessivamente negativa do todo (TARJAN, 1985, p.1).

Ao amortizar as operações de maior custo nas demais, pode-se oferecer uma análise mais próxima da realidade. Além disso, a "análise amortizada difere da análise do caso médio porque a probabilidade não está envolvida. Uma análise amortizada garante o desempenho médio de cada operação no pior caso" (CORMEN *et al.*, 2022, cap. 16). Para amortizar o cálculo de complexidade, pode-se seguir mais de uma metodologia:

a) método de agregação é o mais simples, considera-se o tempo total $T(n)$ para uma sequência de n operações. O custo amortizado é obtido dividindo esse tempo total pelo número de operações, resultando em $\frac{T(n)}{n}$. Isso se aplica mesmo quando a sequência inclui vários tipos de operações (CORMEN *et al.*, 2022, cap. 16.1). A regra pode ser formalizada como na equação

$$C_a = O\left(\frac{\sum_{i=0}^{n} desempenho(i)}{n}\right) \tag{2.8}$$

na qual a complexidade amortizada C_a é obtida somando o desempenho de cada operação i até n e dividindo pelo número total de operações (n);

b) método contábil é utilizado quando diferentes operações possuem diferentes custos. A ideia é que algumas operações irão "providenciar crédito" para outras. Em outras palavras, o trabalho realizado por uma operação serve de crédito para outra (CORMEN *et al.*, 2022, cap. 16.2). Para ilustrar esse método, considera-se o incremento de um número binário. Considere um número que irá incrementar a cada 1, com o valor armazenado em um *array* representando-o em um formato binário. Para cada incremento, é necessário "inverter" o bit menos significativo e o bit a seguir se o atual for virado para 0. A seguir está apresentada a descrição dos passos e um exemplo que pode ser acompanhado no Quadro 1:

   – inicie com o bit menos significativo: este é o bit mais à direita do número binário;
   – verifique o bit: se o bit for 0, mude-o para 1. Nesse caso, você concluiu o processo de incremento;
   – se o bit for 1, mude-o para 0: quando um bit é virado de 1 para 0, você precisa "carregar" 1 para o próximo bit à esquerda;
   – repita o processo para o próximo bit à esquerda: continue esse processo, movendo-se para a esquerda pelos bits, até que não existam mais bits para carregar.

   Se considerar o exemplo de virar bits, o custo de definir um bit pode ser considerado custo duplo, enquanto o de redefinir bits é sem custo. Essa ideia parte do princípio que, para redefinir um bit, ele deve ser inicialmente definido. Ou seja, cria-se um crédito, assim como em instituições financeiras, que poderá ser descontado futuramente em outras operações (CORMEN *et al.*, 2022, cap. 16.2). Assim, pode-se verificar no Quadro 1 que o custo de cada operação varia, mas o custo acumulado é equivalente ao valor do contador, o que significa que a amortizada é $O\,(n)$ (CORMEN *et al.*, 2022, cap 16.2);

Quadro 1 – Incremento de um número binário de 8 bits

Valor do Contador	Valor Binário	Custo Acumulado
0	0$\widehat{0}$0000[0]	0
1	00000[01]	1
2	000001[0]	3
3	00000[1]1	4
4	000010[0]	7
5	00001[01]	8
6	000011[0]	10
7	000[0111]	11
8	000100[0]	15
9	00010[01]	16
10	000101[0]	18
11	0001[011]	19
12	000110[0]	22
13	00011[01]	23
14	000111[0]	25
15	00[01111]	26
16	001000[0]	31

Fonte: O Autor (2024), adaptado de Cormen *et al.* (2022, cap. 16.1).

c) método do físico (ou método potencial) refere-se à utilização de uma função potencial Φ na estrutura de dados. O custo amortizado c'_i de uma operação i é então definido como

$$c'_i = c_i + \Phi\left(D_i\right) - \Phi\left(D_{i-1}\right), \tag{2.9}$$

em que c_i é o custo real da operação, D_i é a estrutura de dados referente à estrutura de dados resultante da operação c_i, $\Phi(D_i)$ é o valor potencial (valor máximo que poderá alcançar) da estrutura de dados após a i-ésima operação e $\Phi(D_{i-1})$ é o potencial da estrutura de dados antes da i-ésima operação (CORMEN *et al.*, 2022, cap 16.3), sendo esse potencial o valor equivalente à complexidade. A ideia é que uma operação pode ser "cobrada" pela diferença de potencial entre as duas operações, ou seja, deve-se ter uma função Φ para uma estrutura de dados na i-ésima operação ($\Phi(D_i)$) que seja sempre maior que $\Phi(D_{i-1})$. Esse potencial extra pode ser pensado como "tempo pré-pago", que pode ser usado para pagar por operações futuras.

2.3 APLICAÇÃO DO CÁLCULO DA COMPLEXIDADE

Uma vez possuindo o conceito da complexidade assintótica, precisa-se de métodos para sua aplicação em códigos de programação. Partindo de uma operação fundamental, pode-se contabilizar quantas vezes ela será executada. Uma quantidade de tempo fixa é necessária para executar cada linha de um código e supõe-se que a execução da k-ésima linha leve um tempo ck, em que ck é uma constante (CORMEN *et al.*, 2022, p.58).

2.3.1 Princípios

Para realizar o cálculo de complexidade, principalmente focando na complexidade pessimista, é necessário o conhecimento de alguns princípios. Esses princípios podem simplificar a quantidade de cálculos a serem realizados.

2.3.1.1 Princípio das componentes

Quando se trata de algoritmos com linguagens de programação, será dificilmente analisado um trecho de código de modo monolítico, mas sim através da separação das suas partes, denominadas componentes. Uma componente de um algoritmo é classificada como conjuntiva quando ela é executada em todas as instâncias da execução desse algoritmo, independentemente do valor de entrada. De modo inverso, são disjuntivas quando apenas algumas são executadas conforme o valor de entrada (TOSCANI; VELOSO, 2012, p.44).

2.3.1.1.1 Componente conjuntiva

Conforme pode ser visto no Algoritmo 1, todas as linhas serão executadas, isso significa que se pode somar o desempenho das partes para obter a complexidade do algoritmo completo.

Algoritmo 1 – Cálculo da média para a exemplificação de componentes conjuntivas

```
1  #include <stdio.h>
2
3  int main() {
4      int numeros[5] = {1, 2, 3, 4, 5};
5      float media = 0;
6      for(int i = 0; i < 5; i++) {
7          media += numeros[i];
8      }
9      media = media / 5;
10     printf("A media dos numeros no array e: %.2f\n", media);
11     return 0;
12 }
```

A seguir estão as descrições de cada passo do algoritmo:

a) linha 4, inicialização do array: é uma operação $O(1)$, pois apenas aloca memória do array com valores constantes;

b) linha 5, declaração da média: é uma operação $O(1)$, pois apenas aloca memória para a variável de média;

c) linhas 6-8, laço para soma dos números: é uma operação $O(n)$, sendo n o número de elementos. Cada elemento é acessado uma vez para realizar a soma;

d) linha 9, cálculo da média: é uma operação $O(1)$, sendo a divisão da soma pelo número de elementos.

Se somar a complexidade de cada componente, o desempenho do algoritmo é $O(1) + O(1) + O(n) + O(1)$.

2.3.1.1.2 Componente disjuntiva

O Algoritmo 2 é um exemplo de código que contém componentes disjuntivos. Ele verifica se um determinado número N fornecido pelo usuário é positivo ou negativo, sendo que imprime todos os números de zero até N caso seja positivo.

Isso significa que se um número é positivo, ele será pelo menos $O(n)$, enquanto se for negativo, será apenas $O(1)$. Assim, se considerar sempre o pior caso, $O(n)$ deve ser a complexidade assintótica, mas não será no caso médio ou na complexidade otimista. Posteriormente, uma maneira de abordar a complexidade de componentes disjuntivas será abordada pelo máximo assintótico na Seção 2.3.1.3.

Algoritmo 2 – Exemplo de código com Componentes Disjuntivas

```
1  #include <stdio.h>
2  int main() {
3      int num;
4      printf("Digite um numero: ");
5      scanf("%d", &num);
6
7      if(num > 0) {
8          for(int i = 0; i < num; i++) {
9              printf("%d ", i);
10         }
11     } else {
12         printf("O numero nao e positivo.\n");
13     }
14     return 0;
15 }
```

2.3.1.2 Princípio da absorção

Essa propriedade é comumente aplicada quando componentes seguirem os seguintes predicados: são conjuntivas, são polinômios sobre a mesma variável e uma componente cresce mais rapidamente que as demais (TOSCANI; VELOSO, 2012, p.51).

Se uma função g cresce mais rapidamente que outra f, então a soma das duas crescerá assintoticamente como g (TOSCANI; VELOSO, 2012, p.52). Para exemplificar, se a função g for $O(n^2)$ e a função f for $O(n)$, isso significa que a soma $O(n^2)+O(n)$ resulta em $O(n^2)$. Como a

função g tem crescimento superior à f, pode-se dizer que a sua complexidade "absorve" a outra, fazendo-se necessário melhorar essa função para melhorar o desempenho de ambas como uma unidade.

2.3.1.3 Máximo assintótico

O princípio da absorção funciona bem para componentes conjuntivas, pois como visto, todas são executadas. No entanto, a situação muda quando se lida com componentes disjuntivas, em que a execução ocorre em um ou outro componente. Nesses casos, pode haver situações em que um componente sempre apresenta uma cota assintótica superior, ou, dependendo do ponto específico da execução, pode haver uma alternância entre qual componente se sobressai. Para lidar com esse último caso, é comum realizar o cálculo do máximo assintótico.

O máximo assintótico (em ordem) é comutativo (TOSCANI; VELOSO, 2012, p.53), ou seja, a ordem dos termos não afeta o resultado. Havendo as funções $f(n)$ e $g(n)$, a ordem assintótica máxima será igual, independentemente da função considerada por primeiro, o que pode ser formalizado com uma equação como

$$\max(f(n), g(n)) = \max(g(n), f(n)), \tag{2.10}$$

significando que a função que cresce mais rapidamente dominará a outra para valores suficientemente grandes de n. Assim, pode-se inferir que se o máximo assintótico é o mesmo independente da ordem, ele é também o máximo para ambas as funções, conforme representado em

$$f = O(\max(f(n), g(n))) \; e \; g = O(\max(f(n), g(n))). \tag{2.11}$$

No entanto, ainda é possível encontrar um máximo assintótico que seja válido, mas que seja demasiado "folgado", ou seja, que para ambas as funções fique distante de maneira superior ao que seria seu verdadeiro máximo. Por exemplo, se f for $O(n)$ e g for $O(n * log(n))$, ambas seriam dominadas por n^2, mas o mais adequado seria afirmar que o máximo assintótico deveria ser $O(n * log(n))$ (TOSCANI; VELOSO, 2012, p.53), podendo ser expresso como a equação

$$\max(f, g) = O(h), \; sempre \; que \; f = O(h) \; ou \; g = O(h). \tag{2.12}$$

Portanto, nos casos em que pudermos estabelecer que, a partir de algum ponto n_0, a função g cresce igual ou mais rapidamente que a função f, o "máximo" de f e g será sempre g, porque g é pelo menos tão grande quanto f para todos os valores de n maiores que n_0. Isso pode ser descrito pela equação

$$\max(f, g) = g, \; sempre \; que \; (\exists n_0 \in \mathbb{N})(\forall n \geq n_0) : f(n) \leq g(n). \tag{2.13}$$

Quando isso não ocorrer, é possível utilizar uma cota assintótica superior, conforme visto anteriormente.

2.3.1.4 Princípio da transformação de entradas

Quando ocorrerem casos em que a execução de uma componente altere o valor de entrada de outra componente (como pode acontecer em laços aninhados), utiliza-se a especialização por transformação de entradas. É possível que dependendo da entrada, um componente tenha uma cota assintótica superior e mesmo se as entradas forem de mesmo tamanho, basta uma função de transformação das entradas ser executada previamente ao componente (TOSCANI; VELOSO, 2012).

2.3.2 Aplicação do cálculo pessimista em equações

Uma vez estabelecidos os princípios, se pode começar a analisar algoritmos, aplicando equações para contabilizar sua complexidade. Em seguida, serão examinados métodos para calcular certos tipos de operações comuns em códigos de programação.

2.3.2.1 Atribuição

A atribuição pode ser tanto uma operação básica de atribuição de valor a uma variável, como pode ser também algo mais complexo, como atualizações de matrizes, grafos, uma expressão ou uma função. Para esses casos mais simples, costuma-se atribuir $\Theta(1)$, ou seja, constante (TOSCANI; VELOSO, 2012, p.55).

No entanto, dependendo da estrutura analisada, pode-se considerar que para cada *byte* ou bloco de memória atribuído de uma variável para outra, deve-se adicionar complexidade, pois essa seria a operação fundamental. Uma lista, por exemplo, no seu pior caso, resultaria em $O(n)$. Se houver ainda uma avaliação de uma expressão ou ainda uma função sendo executada, por exemplo, a inversão da lista, soma-se o custo dessa operação (TOSCANI; VELOSO, 2012, p. 56), conforme a equação

$$C_p[v \leftarrow e] = O(C_p[e] + C_p[\leftarrow_e]) \tag{2.14}$$

na qual a complexidade da atribuição $v \leftarrow e$ é dada pela soma da complexidade da avaliação da expressão $O(C_p[e])$ com a complexidade da atribuição dessa expressão dada por $O(C_p[\leftarrow_e])$.

2.3.2.2 Condicionais

As condicionais (como “se-então-senão” e “se-então”) são usadas para controlar o fluxo de execução. A sua complexidade pessimista é geralmente o máximo entre a complexidade das duas ramificações.

Para condicionais simples (se-então), basta avaliar a complexidade da expressão da condicional e a complexidade do bloco de código caso a condição seja verdadeira, afinal, se considerar o pior caso, ela será verdadeira (TOSCANI; VELOSO, 2012, p.57), podendo ser descrito

conforme equação

$$C_p[se\ b\ entao\ S\ fimse] = O(C_p[b] + C_p[S]); \tag{2.15}$$

para as condicionais completas (se-então-senão), em que a lógica é muito similar, a diferença é relativa aos casos em que o condicional for falso, que pode ter complexidade maior. Nesse caso, deve-se optar pela utilização do pior caso, conforme apresentado na equação

$$C_p[se\ b\ entao\ S\ senao\ T\ fimse] = O(C_p[b] + \max(C_p[S], C_p[T])), \tag{2.16}$$

em que se considera o máximo assintótico entre os dois blocos **S** e **T** somado ao custo da avaliação da condicional.

2.3.2.3 Composição

A composição se trata de uma estrutura simples, formada pela sequência de componentes (TOSCANI; VELOSO, 2012, p.60). Portanto, a sua complexidade da sequência é a soma da complexidade de cada componente, podendo ser descrita como

$$C_p[S;T] = O(C_p[S]) + O(C_p[T]). \tag{2.17}$$

Se denomina esse cenário de complexidade de composição com preservação assintótica de tamanho (TOSCANI; VELOSO, 2012, p.61).

No entanto, quando uma componente realiza alteração na entrada de dados d da próxima componente, é necessário considerar esse cenário, que seria a complexidade de composição sem preservação assintótica de tamanho. No entanto, para facilitar, é interessante definir uma regra geral, que funciona tanto para composição com e sem preservação de tamanho. Essa regra geral foi providenciada por Toscani e Veloso (2012, p.62), sendo exemplificada na equação

$$C_p[S;T] = (C_p[S](n) + C_p[T](S(n))), \tag{2.18}$$

em que dada a sequência $S;T$, pode-se considerar $S(n)$ o tamanho máximo da saída de S para entradas com tamanho até n.

2.3.2.4 Iteração incondicional

A iteração incondicional se trata de operações executadas de maneira repetidas, em que, de maneira geral, há previsibilidade, seguindo comumente a estrutura "para-de-até-faça" (TOSCANI; VELOSO, 2012, pg. 62). A equação

$$C_p[para\ i\ de\ j\ ate\ m\ faca\ S\ fimpara](n) = O\left(\sum_{i=j(n)}^{m(n)} C_p[S]\left(s^{i-j(n)}(n)\right)\right) \tag{2.19}$$

descreve o custo pessimista de um laço que executa uma operação S para um índice i que varia de j até m, em termos do tamanho da entrada n. O custo de tempo é dado pela soma dos custos pessimistas da operação S para cada valor de i de $j(n)$ até $m(n)$.

A operação S e a transformação $s^{i-j(n)}$ podem variar dependendo do algoritmo ou operação específica que está sendo analisada. O custo de tempo dessa operação S pode variar dependendo de quão complexa é (como exemplo, ela pode inclusive ser outra estrutura de controle de laços, caracterizando laços aninhados). A transformação de $s^{i-j(n)}$ pode variar dependendo do algoritmo ou da operação analisada. Em um algoritmo de busca binária, a transformação pode ser uma divisão por 2 ($s^{i-j(n)} = n/2^i$), pois o tamanho da entrada é reduzido pela metade a cada iteração.

2.3.2.5 Iteração condicional

As iterações condicionais se referem às operações executadas de maneiras repetidas, em que, de maneira geral, não há previsibilidade de encerramento, comumente associadas à estrutura "enquanto-faça". Em outras palavras, existe uma condição para a qual o laço deve continuar sendo executado (TOSCANI; VELOSO, 2012, p.69).

Conforme Toscani e Veloso (2012, p.74), a complexidade temporal de um laço "enquanto-faça" é demonstrada na equação

$$C_p\left[\text{enquanto } b \text{ faca } S \text{ fimenquanto}\right] = O\Bigg(C_p[b]\left(s^{h(n)}(n)\right) + \sum_{i=0}^{h(n)-1}\left[C_p[b]\left(s^i(n)\right) + C_p[S]\left(s^i(n)\right)\right]\Bigg) \tag{2.20}$$

em que é possível verificar que a complexidade é determinada pelo tempo que leva para avaliar a condição b do laço e executar a instrução S. O laço é executado $h(n)$ vezes antes de parar. Para cada iteração i (de 0 a $h(n) - 1$), o tamanho da entrada para b e S é $s^i(n)$. A complexidade temporal para cada iteração inclui o tempo para avaliar a condição b ($C_p[b]\,(s^i(n))$) e o tempo para executar a instrução S ($C_p[S]\,(s^i(n))$). A complexidade temporal total do laço é a soma das complexidades temporais para todas as $h(n)$ iterações, mais a complexidade temporal de avaliar a condição b pela $h(n)$-ésima vez ($C_p[b]\left(s^{h(n)}(n)\right)$). Essa complexidade temporal total é representada na notação Big O, indicando o cenário de pior caso.

2.4 PROBLEMA DA DECISÃO

Na virada do século XIX para o século XX, Hilbert (1900), um dos matemáticos mais influentes da sua época, realizou um discurso incentivando seus companheiros de profissão e jovens aspirantes a resolverem certos problemas no campo da matemática, como objetivo para este novo século[1,2]. Ele propunha esses problemas por crer que a matemática deveria ser axio-

[1] Discurso proferido em *Sorbonne* na conferência do Congresso Internacional de Matemáticos, em 8 de agosto de 1900.

[2] O seu discurso precedeu a publicação dos problemas.

mática, completa em si. Posteriormente, ele solicitou "[...] um processo geral para determinar a demonstrabilidade de declarações arbitrárias na lógica matemática" (PETZOLD, 2008, p.ix *apud* HILBERT; ACKERMANN,1928). Encontrar esse "processo geral" ficou conhecido como *Entscheidungsproblem*, sendo descrito a seguir:

> [...] encontrar um algoritmo que recebe como entrada a descrição de uma linguagem formal e uma sentença nesta linguagem e tem como saída "verdadeiro" ou "falso" dependendo se a sentença da entrada é verdadeira ou falsa (DIVERIO; MENEZES, 2011, p. 23).

Esse problema é um dos pilares da área de ciência da computação e quando se trata de analisar estaticamente um código, pode-se levantar questões relativas, afinal, Turing e Church chegaram à conclusão que não existe solução para o problema da decisão, isso é, não há como saber se existe uma solução para qualquer problema (DIVERIO; MENEZES, 2011, p. 24).

2.4.1 Máquina de Turing

Turing (1936, p.230) mostrou que o problema Hilbertiano *Entscheidungsproblem* não tem solução. Ele chegou a essa conclusão a partir de números computáveis, partindo do princípio de que são enumeráveis. Para isso, ele assume que os números computáveis são, portanto, finitos, e um computador (na época, uma pessoa) teria memória limitada.

Partindo desse preceito, ele criou uma máquina hipotética análoga ao computador humano, ou seja, que poderia executar as mesmas atividades de maneira metódica. Ele define como seria essa máquina em termos que poderiam substituir algo real por outro imaginário, um análogo, como, por exemplo, ao invés de um papel, uma fita. Essa fita seria dividida em quadrados, sendo que um deles por vez poderia ser "escaneado" por um cabeçote e o símbolo escrito dentro desse quadrado por consequência seria o símbolo "escaneado". O comportamento da máquina é definido por uma série finita de configurações, chamadas de "m-configurações" e a memória da máquina se restringe à sua m-configuração atual (que pode ser alterada) pelo símbolo escaneado, o qual é o único que a máquina tem conhecimento (TURING, 1936, p.231).

Essa configuração é definida por um conjunto de instruções fornecidas à máquina, especificando como ela deve operar. Essas instruções determinam o estado inicial da máquina, as condições para transições entre estados e as ações a serem executadas, como mover o cabeçote de leitura/gravação ou alterar um símbolo na fita durante cada estado. Esse conjunto de regras é denominado Descrição Simbólica (*Symbolic Description*, S.D., em inglês) (TURING, 1936, p.240).

Em seguida, Turing apresenta o conceito de máquina universal.

> É possível inventar uma única máquina que pode ser usada para calcular qualquer sequência computável. Se esta máquina U for fornecida com uma fita no início da qual está escrito o S.D. de alguma máquina de computação M, então U calculará a mesma sequência que M (TURING, 1936, p.241).

Mesmo com algumas limitações, Turing

> [...] demonstrou a generalidade da computação ao mostrar que uma única máquina universal pode ser adequadamente programada para realizar a operação de qualquer máquina de computação (PETZOLD, 2008, p.161).

Partindo do pressuposto que números computáveis são finitos e da definição da máquina universal, não seria possível, portanto, definir um S.D., e por consequência um número descritivo (D.N.) enumerável, de uma máquina que tende ao infinito, como máquinas livres de círculo, pois a sua definição tende ao infinito. Se não se pode definir uma máquina assim, significa então que as máquinas são finitas e, portanto, enumeráveis.

> [...] o problema de enumerar sequências computáveis é equivalente ao problema de descobrir se um determinado número é o D.N. de uma máquina livre de círculos, e não há um processo geral para fazer isso em um número finito de paços. Na verdade, aplicando corretamente o argumento do processo diagonal, se pode mostrar que não pode haver nenhum processo geral desse tipo (TURING, 1936, p.246).

Posteriormente, Turing usa da redução ao absurdo para provar que uma máquina capaz de determinar o resultado antes da sua execução não pode existir, resolvendo então *Entscheidungsproblem*. Ele assume que uma máquina hipotética com determinada configuração existe e conseguiria determinar se qualquer máquina de Turing imprime 0 infinitamente. Se ela existisse, então se poderia construir um processo para determinar se uma máquina de Turing imprime "1" infinitamente. Isso é porque imprimir "1" infinitamente equivale ao complemento de imprimir "0" infinitamente. Se for possível determinar se uma máquina de Turing imprime "0" ou "1" infinitamente, então pode-se determinar se uma máquina de Turing é livre de círculos. No entanto, já foi provado que é impossível determinar se uma máquina de Turing é livre de círculos. Portanto, a existência da máquina assim leva a uma contradição, o que significa que a máquina não pode existir (TURING, 1936, p.248).

Pela Máquina Universal de Turing, não seria possível inferir se uma função com uma determinada entrada encontraria um resultado (ou pararia a sua execução em algum momento). Isso significa que, como regra universal, qualquer computador não poderia determinar antes de executar determinada função se essa função irá terminar ou não.

A Tese de Church, que estabelece que "a capacidade de computação representada pela Máquina de Turing é o limite máximo que pode ser atingido por qualquer dispositivo de computação" (DIVERIO; MENEZES, 2011, p.175), foi a base para essa conclusão. Isso é corroborado por várias tentativas de desenvolver máquinas universais, como a Máquina Norma e a Máquina de Post, que produziram conceitos equivalentes aos de Turing (DIVERIO; MENEZES, 2011, p.175).

Esse problema impacta a capacidade de calcular a complexidade de uma função qualquer. Afinal, se não é possível determinar se uma função irá parar de executar, como se pode

avaliar a sua complexidade? Para realizar essa análise de complexidade, precisa-se determinar limites para ela. Tais limites serão discutidos mais adiante na proposta de solução.

3 ESTADO DA ARTE

Neste capítulo são evidenciadas as aplicações que se relacionam com o objeto de estudo deste trabalho, assim como a avaliação das suas contribuições e limitações.

3.1 FERRAMENTAS DE ANÁLISE DE CÓDIGO

Analisadores estáticos de código disponibilizados de maneira comercial são muito similares entre si. A maioria realiza a análise por meio de uma base de dados com regras e casos comuns. Além disso, quando há análise de complexidade, ela se limita à complexidade ciclomática ou relacionada à legibilidade.

Dentre eles está o Qodana[1] , que realiza as análises por meio de uma base de dados que está disponibilizada em um portal chamado Inspectopedia [2] , que contém os problemas mais recorrentes e comuns em um projeto de *software* em que uma determinada implementação não é ótima ou recomendada.

Outras ferramentas similares são o Embold [3] e o Sonar [4] , tanto a versão autônoma (*stand-alone*) denominada SonarQube como a cloud (SonarCloud). Essas ferramentas também realizam a análise do código, variando de funcionalidades entre si, como a presença de indicadores, detecção de segredos, integração para *deploy* e recomendações geradas por inteligência artificial.

No entanto, esse trabalho não visa analisar a legibilidade, clareza ou padrões de codificação, mas estimar um desempenho pelo código-fonte.

3.2 ANALISADORES DE DESEMPENHO

Alguns projetos foram criados para realizar a análise de desempenho no tempo, considerando ou o código-fonte ou algum tipo de representação intermediária, como o *bytecode* Java ou até mesmo com a utilização de perfiladores de código. O cenário mais comum para este tipo de ferramenta ainda se encontra focado no ambiente acadêmico, seja como objeto de pesquisa ou na criação de uma ferramenta para utilização educacional.

---

[1] JETBRAINS. **Qodana: Static Code Analysis Tool by JetBrains**. 2024. Disponível em: <https://www.jetbrains.com/qodana/>. Acesso em: 06 abr. 2024.

[2] JETBRAINS. **Inspectopedia Documentation**. 2024. Disponível em: <https://www.jetbrains.com/help/inspectopedia/WhatIsNewInspections.html>. Acesso em: 06 abr. 2024.

[3] EMBOLD. **Embold | Static Code Analysis Platform**. 2024. Disponível em: <https://embold.io/>. Acesso em: 06 abr. 2024.

[4] SONAR. **Code Quality, Security and Static Analysis Tool with SonarQube | Sonar**. 2024. Disponível em: <https://www.sonarsource.com/products/sonarqube/>. Acesso em: 06 abr. 2024.

3.2.1 MaLiJan — *Maximum Likelihood Java Bytecode Analyzer*

A ferramenta *Maximum Likelihood Java Bytecode Analyzer (MaLiJan)* (LAUBE; NEBEL, 2010) realiza a análise de complexidade pela execução de distribuições de entrada arbitrárias, avaliando o caso médio de programas Java *Bytecode*. Os resultados obtidos podem ser expressos como número real de instruções *bytecode* executadas ou como medições de custo abstratas, como o número de comparações em algoritmos de ordenação (fornecidas pelo usuário por anotações de código).

Ele, em alguns aspectos, se aproxima do objetivo deste trabalho, mas se distancia no momento que analisa código *bytecode*, uma vez que deixa de avaliar o código-fonte em si. Além disso, ele não retorna a complexidade assintótica, mas um contador de instruções.

3.2.2 COSTA — *COSt and Termination Analyzer for Java Bytecode*

O *COSt and Termination Analyzer for Java Bytecode (COStA)* (ALBERT *et al.*, 2008) é um protótipo de pesquisa que pode analisar automaticamente programas e determinar os detalhes de custo e término de programas em *bytecode* Java. O sistema tenta limitar o consumo de recursos do programa em relação ao modelo de custo especificado, após receber como entradas um programa em *bytecode* e um modelo de custo selecionado de uma lista de descrições de recursos. O número de instruções em *bytecode* executadas, a quantidade de memória alocada no *heap* e o número de eventos faturáveis (como enviar uma mensagem de texto em um dispositivo móvel) que o programa executa são apenas algumas das ideias não triviais de recursos que o COStA oferece.

Embora fundamentada em diversos artigos, essa aplicação, infelizmente, se encontra inacessível, podendo ser encontradas referências da sua existência apenas em *cache* de repositórios e os artigos publicados. Além disso, o foco parece ser em uso de recursos (com foco monetário, conforme descrito nos eventos faturáveis), ao invés da complexidade computacional.

3.2.3 Avaliador automático para ferramentas *IGED* e *MOCCA*

Em 2014 foi apresentada uma ferramenta para analisar a complexidade de código em ambientes de aprendizagem (COSTA *et al.*, 2014) chamadas Interpretador Gráfico de Estrutura de Dados (IGED) e *Measurement of Complexity of an Certain Algorithm (MOCCA)* em um evento denominado *Computer on the Beach*, realizado pela Universidade do Vale do Itajaí (UNIVALI).

Essas duas ferramentas foram criadas com o intuito de auxiliar em ambiente educacional para determinar se uma implementação do código de um aluno corresponde às expectativas. No cenário apresentado no artigo, existe um ambiente de programação em que uma determinada atividade é apresentada e o aluno deve fornecer uma solução. Essa solução pode estar correta do ponto de vista do resultado, mas não utiliza necessariamente as técnicas solicitadas. Um caso disso é a ordenação de listas. O aluno pode resolver um problema utilizando *bubble-*

sort enquanto o problema solicitava utilizando *merge-sort*. Isso significa que se a ferramenta analisar apenas o resultado, pode ocorrer em um falso-positivo. Uma vez obtida a classificação assintótica, é possível validar se a implementação está correta, do ponto de vista da sua complexidade.

Não distante, existem ferramentas denominadas *online-judge* nas quais a validação ocorre perante o tempo de execução, existindo um limite máximo para a execução. No entanto, para isso é necessário executar o código providenciado pelo aluno e analisá-lo perante o tempo, o que pode variar conforme a máquina.

Os autores apresentam o "Avaliador de Eficiência", cuja função é de "gerar várias instâncias aleatórias de tamanhos n como entrada para o algoritmo e armazenar a quantidade de comandos básicos contabilizados pelo contador de comandos da ferramenta, alimentando a Tabela de Eficiência" (COSTA *et al.*, 2014). Esse contador foi gerado com uma ferramenta de Python que realiza a análise de perfilamento do programa.

A Tabela de Eficiência contém informações como a dimensão dos dados, a contagem dos passos e o pico máximo de alocação de memória. A partir desses dados, se constrói a função de custo do processamento $T(n)$ de um algoritmo. Assim, o valor pode ser plotado em um gráfico e visualmente comparado com outros algoritmos pré-cadastrados.

Para esse propósito, é definido um Fator de Classificação (FC), que estabelece um valor numérico para classificar a eficiência de um algoritmo. Cada tipo de algoritmo (linear, quadrático, dentre outros) possui o seu próprio fator de classificação, demonstrando o seu comportamento.

Após obter os FCs, o algoritmo é analisado, calculando o seu Coeficiente de Classificação (CC). Durante esse processo de cálculo, a função de custo do algoritmo (obtida na tabela de eficiência) é examinada em razão da sua derivada primeira usando a Interpolação Polinomial de Lagrange. O valor resultante é então comparado com os FCs conhecidos, permitindo assim uma classificação assintótica ao identificar qual FC é mais próximo do CC encontrado.

Embora exista a utilização de um perfilador para a realização da contagem de passos, o restante do trabalho se assemelha com o objetivo deste. Uma questão que parece um limitador é que ele necessita de uma base de algoritmos pré-definidos para ser realizada uma comparação, visto que os algoritmos possuem foco de avaliação perante uma instrução prévia ao desenvolvedor/aluno, algo que não ocorre no cenário estabelecido para esse trabalho.

3.2.4 ANAC — Uma Ferramenta para a Automatização da Análise da Complexidade de Algoritmos

O artigo que propõe o Analisador de Complexidade (ANAC) (BARBOSA, 2001) apresenta o desenvolvimento de um protótipo de uma ferramenta cujo objetivo é auxiliar no ensino

e análise da complexidade de algoritmos, fornecendo equações de complexidade para o caso médio e o pior caso.

A metodologia utilizada para analisar os algoritmos envolve a implementação da metodologia de cálculo de complexidade para estruturas algorítmicas proposta por Toscani e Veloso (1990). Ela abrange as principais estruturas, como atribuição, sequência, condicional, iteração e iteração condicional, direcionada principalmente para a análise no pior caso. Ele analisa algoritmos não recursivos escritos em uma linguagem semelhante a Pascal, identificando as estruturas algorítmicas presentes e construindo equações de complexidade simplificadas. O resultado da análise é apresentado em ordem assintótica de complexidade.

Na etapa de construção do sistema, foi realizada a definição e construção de um analisador léxico-semântico, utilizando uma estratégia *top-down* com o método de análise preditiva-tabular. Essa abordagem envolve a análise da estrutura do código-fonte do algoritmo e criação de uma *AST* para identificar as diferentes construções algorítmicas e, a partir disso, construir as equações de complexidade associadas.

Algumas questões de restrições foram notadas, como nos laços condicionais ou iteração, cujo usuário é requisitado a inserir valores relativos à complexidade, tornando o processo, até certo ponto, manual.

3.2.5 Análise de *Big-O* automatizada para programas Java

Na *International Conference on Nascent Technologies in Engineering (ICNTE)* 2017 foi apresentado um algoritmo (VAZ *et al.*, 2017) para analisar estaticamente um código Java simples (sem ramificações, sem suporte a recursões, de nível introdutório para um aluno de Ciência da Computação).

Esse algoritmo mescla técnicas de compiladores para a realização do cálculo, realizando o *parsing* do código e armazenando palavras-chave e símbolos, identificando laços de repetição, lógica de programação modular (implementado em Java por funções/métodos), crescimento logarítmico e crescimento polinomial.

O trabalho se limita por analisar apenas a linguagem Java, uma quantidade restrita de palavras-chave e por não possibilitar análise de código ramificado em diversos arquivos. No entanto, ele consegue realizar a análise de loops aninhados, quando esses têm a mesma taxa de crescimento.

Destaca-se também a estrutura de dados utilizada, consistindo principalmente pelos itens listados a seguir:

a) tabela de palavras-chave (*keyword table*) armazena as palavras-chave mais significativas do código, juntamente com a ordem de crescimento correspondente para loops específicos. Por exemplo, a ordem de crescimento linear (n), quadrática (n^2)

ou logarítmica ($\log_2 n$) para loops;

b) tabela de funções (*function table*) registra as funções principais do programa, como *main* e *sort*, e calcula a complexidade de cada função com base na complexidade dos loops identificados na Tabela de Palavras-chave. A complexidade total da função é determinada considerando quais loops estão contidos nela;

c) lista de Intervalos (*interval list*) é utilizada para capturar o escopo das estruturas e processá-las com base na ordem de execução. Essa estrutura é implementada para organizar e representar informações sobre o código, como linhas de início e fim de loops, número de declarações, entre outros aspectos.

3.3 TENDÊNCIAS E INOVAÇÕES: *LARGE LANGUAGE MODELS*

Os modelos de linguagem de grande escala, conhecidos como LLMs, surgiram com o propósito de processar e gerar texto com comunicação coerente e de generalizar para múltiplas tarefas. Assim, as atividades normalmente laboriosas antes delegadas às pessoas podem ser facilmente executadas por tais ferramentas. Dentre elas estão interpretar textos com contexto e a resolução de problemas matemáticos (NAVEED *et al.*, 2024).

Há de se presumir que analisar códigos de programação é uma de suas capacidades, o que realmente se reflete em ferramentas como o Copilot[5] . Por possuir capacidade de resolução de problemas matemáticos, pode-se instruir uma ferramenta para que não somente analise o código, mas também calcule sua complexidade assintótica mediante *prompt engineering*, na qual as instruções de linguagem natural são adaptadas à ferramenta e aperfeiçoadas para atingir um resultado com a LLM (NAVEED *et al.*, 2024).

Para esse projeto, é possível a utilização de uma LLM (como o Ollama[6] , Gemini[7] e o Copilot[8]), em que o código pode ser fornecido juntamente da instrução para analisar a complexidade assintótica. Foi verificado que, embora seja possível, essa estratégia resulta em custos monetários, devido às taxas de utilização de *Application Programming Interface (API)* em ferramentas proprietárias. Mesmo se executado localmente, requer *hardware* específico para execução da ferramenta (uma *Graphics Processing Unit (GPU)* de última geração ou uma *Neural Processing Unit (NPU))*. Utilizar ferramentas de código aberto resultaria em uma ferramenta grande demais para ser incorporada a uma Integrated Development Environment (IDE), visto que os modelos normalmente alcançam gigabytes. Elas poderiam ser utilizadas fornecendo uma API, mas que também acarretam custos de manutenção e segurança de servidor.

[5] MICROSOFT. **Microsoft Copilot | IA da Microsoft**. 2024. Disponível em: <https://www.microsoft.com/pt-br/microsoft-copilot>. Acesso em: 10 jun. 2024.

[6] OLLAMA. **Ollama**. 2024. Disponível em: <https://ollama.com/>. Acesso em: 01 jul. 2024.

[7] TEAM, G. *et al.* **Gemini: A Family of Highly Capable Multimodal Models**. 2024. Disponível em: <https://arxiv.org/abs/2312.11805>. Acesso em: 10 jun. 2024.

[8] MICROSOFT. **Microsoft Copilot | IA da Microsoft**. 2024. Disponível em: <https://www.microsoft.com/pt-br/microsoft-copilot>. Acesso em: 10 jun. 2024.

Em testes realizados com as ferramentas acima citadas, foi evidenciada a necessidade de um *prompt* específico para cada ferramenta para ser retornado apenas o *Big-O* sem textos explicativos. Isso implica em mais *tokens* e em um maior custo computacional para a geração do resultado. Para fins de redução do custo computacional, a LLM pode ser utilizada em casos em que não seja possível identificar a complexidade mediante um algoritmo, delegando para uma aplicação local ou remota.

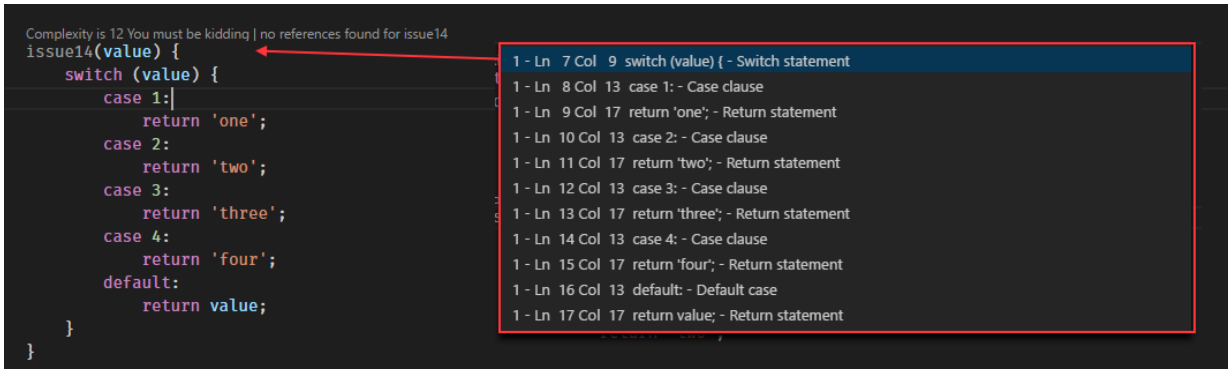
3.4 EXTENSÕES PARA VS CODE

Existem algumas extensões desenvolvidas para o VS Code visando fornecer informações sobre o código-fonte. No entanto, aquelas que se propõem a analisar a complexidade de desempenho ao longo do tempo utilizam inteligência artificial generativa, necessitando de uma chave de API de uma ferramenta paga. Outras ferramentas se limitam a fazer a análise de complexidade do código, como a complexidade ciclomática, *code smells* e outros.

3.4.1 CodeMetrics

A extensão *CodeMetrics*[9] cria uma AST a partir do código-fonte e percorre cada nó dessa árvore. Dependendo do tipo do nó, cria uma entrada. Essa entrada contém tudo o que é necessário para uso posterior, como uma representação textual, incremento de complexidade, nós filhos, entre outros. Ao final, os nodos são contabilizados, sendo fornecido um número representativo à complexidade do código conforme a Figura 4. Isso pode servir como base para o desenvolvimento da extensão de análise de desempenho.

Figura 4 – Captura de tela do CodeMetrics



Fonte: CodeMetrics (2024)

[9] KISS, T. **CodeMetrics**. 2024. Disponível em: <https://github.com/kisstkondoros/codemetrics>. Acesso em: 09 abr. 2024.

3.4.2 *CodeScene*

A extensão *CodeScene*[10] também realiza a análise de código-fonte, com o foco em providenciar informações e conhecimentos para o desenvolvedor, de maneira que a qualidade de seu código seja melhorada. Dentre as análises realizadas estão a Complexidade Aninhada (instruções condicionais ou *loops* altamente aninhados), *Bumpy Road* (vários pedaços de complexidade aninhados), funções complexas (medidas como alta complexidade ciclomática), funções com muitos argumentos, funções muito longas, entre outras. Além disso, na fase beta também estão disponíveis sugestões de código via inteligência artificial.

Assim como *CodeMetrics*, o *CodeScene* providencia a informação da qualidade do código pela API de *CodeLens*, que está descrita no Capítulo 4.

3.4.3 *Gitlens*

A extensão *Gitlens*[11] não realiza a análise de código, mas providencia informações relevantes ao desenvolvedor relacionadas ao controle de versionamento, como autor, histórico de *commits*, data da última alteração, entre outros aspectos. Isso evidencia que o uso de anotações de código é muito útil como ferramenta para o desenvolvedor, visto a popularidade de tal extensão, com mais de 31 milhões de downloads até a data de escrita deste trabalho.

3.5 ANÁLISE CONSOLIDADA E LACUNAS EXISTENTES

Ao longo deste capítulo, foram apresentadas diversas ferramentas e abordagens que tratam, de maneira direta ou indireta, a análise de código-fonte e complexidade computacional. Embora cada uma ofereça contribuições relevantes, também apresentam limitações que justificam a continuidade de investigações e propostas. O Quadro 2 apresenta um comparativo com as principais características dessas tecnologias.

3.5.1 Aspectos positivos e negativos das ferramentas

Com base no Quadro 2, destacam-se os seguintes pontos:

a) Ferramentas comerciais, como Qodana, Sonar e Embold, oferecem uma gama ampla de funcionalidades, mas com foco em legibilidade e padrões de codificação, não atendendo à necessidade de análise de desempenho baseada em complexidade computacional.

[10] CODESCENE. **CodeScene**. 2024. Disponível em: <https://github.com/codescene-oss/codescene-vscode>. Acesso em: 09 abr. 2024.

[11] GITKRAKEN. **GitLens**. 2024. Disponível em: <https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens>. Acesso em: 12 abr. 2024.

Quadro 2 – Comparação de ferramentas

Ferramenta	**Análise de Complexidade**	**Código-Fonte/Bytecode**	**Aplicação Atual**
Qodana/Embold/Sonar	Ciclomática	Código-Fonte	Industrial/Comercial
MaLiJan	Contador de Instruções	Bytecode Java	Acadêmica
COStA	Custo e Término	Bytecode Java	Acadêmica
IGED/MOCCA	Passos e Eficiência	Código-Fonte	Educacional
ANAC	Assintótica	Código-Fonte (Pascal)	Educacional
Big-O Java	Loops Simples	Código-Fonte Java	Educacional
CodeMetrics	Ciclomática	Código-Fonte	Ferramenta VS Code
CodeScene	Ciclomática e Qualidade	Código-Fonte	Ferramenta VS Code
GitLens	Controle de Versão	N/A	Ferramenta VS Code

Fonte: O Autor (2024).

b) Projetos acadêmicos, como MaLiJan, COStA e ANAC, possuem alta especialização, mas enfrentam limitações quanto à acessibilidade, aplicabilidade em outros ambientes e suporte a linguagens modernas.

c) Ferramentas educacionais, como IGED/MOCCA e Big-O Java, são úteis para treinamento, mas frequentemente requerem interações manuais ou apresentam suporte limitado a cenários reais.

d) Extensões do VS Code, como CodeMetrics e CodeScene, focam na qualidade do código e complexidade ciclomática, enquanto GitLens explora informações de versionamento, demonstrando uma oportunidade para integrar análises de complexidade de desempenho no ambiente de desenvolvimento.

Esse projeto cobre algumas lacunas identificadas no estado da arte, iniciando pela falta de análise de desempenho focada em complexidade computacional, uma vez que muitas ferramentas priorizam legibilidade e padrões de codificação sem oferecer uma avaliação detalhada do desempenho dos algoritmos. Além disso, a acessibilidade e usabilidade em ambientes modernos, ao ser desenvolvida como uma extensão para o Visual Studio Code, facilitam o uso por desenvolvedores em diversos contextos, ainda mais quando disponibilizado gratuitamente, sem a necessidade de contatar instituições de ensino para utilização de ferramentas proprietárias ou limitada aos cenários educacionais.

4 DESENVOLVIMENTO DA EXTENSÃO

Uma extensão[1] de IDE, especificamente no VSCode, permite que durante o desenvolvimento seja possível obter *feedback* em tempo de desenvolvimento e/ou compilação sobre a implementação de uma solução. Dentre esses *feedbacks*, podem-se citar os *linters*, "erros de compilação", sugestões de código e outros aspectos. Com isso, o desenvolvedor pode ficar seguro de que sua aplicação tem salvaguardas antes mesmo de colocá-la no ambiente de execução. Um ponto em que se verifica uma limitação é em relação à análise de desempenho computacional, em que as extensões presentes se limitam a verificar perante casos conhecidos e comuns, mas não apresentam se o código, para uma entrada suficientemente grande, desempenha bem ou mal.

Além disso, *Quality Gates* (SONAR, 2024b) como o SonarQube/SonarCloud também não consideram esse tipo de análise nas suas ferramentas, mesmo se considerar a adição de LLMs como ferramentas para revisão de código. Essa funcionalidade é um indicador de se o código atende critérios de qualidade e pode servir de condição para um código ser incorporado oficialmente a um produto.

4.1 PLANEJAMENTO DA SOLUÇÃO

Para atender às necessidades do projeto, foram descritos requisitos funcionais e não funcionais e, a partir desses, foram criados casos de uso para embasar o desenvolvimento.

4.1.1 Requisitos

Foram criados requisitos funcionais e não funcionais para atender as necessidades descritas, sendo listados a seguir.

4.1.1.1 Requisitos funcionais

Os requisitos a seguir constam todas as regras de negócio necessárias para atender os casos de uso:

a) **instalar extensão:**
   - **prioridade:** alta;
   - **descrição:** o sistema deve permitir que o usuário instale a extensão no VSCode. A extensão deve ser criada e disponibilizada com base na *Software Development Kit (SDK)* descrita na Seção 4.1.4.1;

[1] EXTENSãO Big O. 2024. Disponível em: <https://github.com/francopan/code-performance-analyzer-ext>. Acesso em: 18 nov. 2024.

- **pré-condições:** o usuário tem o VSCode instalado;
- **pós-condições:** a extensão está instalada e pronta para ser usada no VSCode. Tal uso é evidenciado pela função *activate*;

b) **abrir arquivo no VSCode:**

- **prioridade:** alta;
- **descrição:** o sistema deve permitir que o usuário abra um arquivo no VSCode;
- **pré-condições:** o usuário tem um arquivo (código-fonte) que deseja abrir;
- **pós-condições:** o arquivo está aberto no VSCode. Extensão adiciona anotações para a análise de função acima de cada função;

c) **selecionar função:**

- **prioridade:** alta;
- **descrição:** o sistema deve permitir que o usuário selecione uma função no arquivo aberto para a análise;
- **pré-condições:** o usuário tem um arquivo aberto no VSCode que contém funções anotadas;
- **pós-condições:** a função está selecionada e aciona gatilho para a análise;

d) **executar análise *Big-O*:**

- **prioridade:** alta;
- **descrição:** o sistema deve executar uma análise *Big-O* na função selecionada pelo usuário;
- **pré-condições:** o usuário selecionou uma função para a análise;
- **pós-condições:** a análise *Big-O* é executada na função selecionada e exibe o valor ao lado da anotação;

e) **executar análise *Big-O* via LLM:**

- **prioridade:** alta;
- **descrição:** o sistema deve permitir a execução da análise de complexidade *Big-O* via LLM;
- **pré-condições:** a análise de complexidade *Big-O* foi iniciada pelo usuário;
- **pós-condições:** a análise de complexidade *Big-O* é concluída e o resultado é enviado de volta para a extensão;

f) **tratar erro de análise:**

- **prioridade:** alta;
- **descrição:** o sistema deve tratar erros que ocorrem durante a análise de complexidade *Big-O*;
- **pré-condições:** ocorreu um erro durante a análise de complexidade *Big-O*;

- **pós-condições:** o erro é tratado e o usuário é informado;

g) **tratar erro de API:**

- **prioridade:** alta;
- **descrição:** o sistema deve tratar erros que ocorrem durante a execução da análise *Big-O* via LLM;
- **pré-condições:** ocorreu um erro durante a execução da análise de complexidade *Big-O* via LLM;
- **pós-condições:** o erro é tratado e a extensão é informada;

h) **exibir resultados:**

- **prioridade:** alta;
- **descrição:** o sistema deve exibir os resultados da análise para o usuário;
- **pré-condições:** a análise da função foi concluída;
- **pós-condições:** exibe o resultado da análise anotado pelo *code-lens* e em uma janela própria.

4.1.1.2 Requisitos não funcionais

A seguir, estão listados os requisitos não funcionais, ou seja, que dizem respeito a questões que afetam diretamente o produto, mas que não fazem parte da regra de negócio:

a) **tempo de análise:**

- **prioridade:** alta;
- **tipo:** desempenho;
- **descrição:** o sistema deve poder executar a análise *Big-O* em um tempo configurado pelo usuário;

b) **múltiplas solicitações:**

- **prioridade:** alta;
- **tipo:** desempenho;
- **descrição:** o sistema deve lidar com múltiplas solicitações de análise simultaneamente;

c) **segurança na análise*Big-O*:**

- **prioridade:** alta;
- **tipo:** segurança;
- **descrição:** o sistema deve garantir que a análise *Big-O* seja realizada de maneira segura, sem causar danos ao sistema do usuário;

d) **precisão dos resultados:**

- **prioridade:** alta;
- **tipo:** qualidade;
- **descrição:** o sistema deve fornecer resultados precisos de análise;

e) **facilidade de uso:**

- **prioridade:** alta;
- **tipo:** qualidade;
- **descrição:** o sistema deve ser fácil de usar, com uma interface de usuário intuitiva;

f) **anotar código ao nível de função:**

- **prioridade:** alta;
- **tipo:** sistema/técnico;
- **descrição:** o sistema deve conseguir anotar o código-fonte ao nível de função para facilitar a análise;
- **pré-condições:** código da função;
- **pós-condições:** código anotado;

g) **fazer *parsing* do código e transformar em AST:**

- **prioridade:** alta;
- **tipo:** sistema/técnico;
- **descrição:** o sistema deve conseguir fazer *parsing* do código-fonte e transformá-lo em uma AST para a análise;
- **pré-condições:** código de Função;
- **pós-condições:** estrutura da AST;

h) **analisar complexidade localmente:**

- **prioridade:** alta;
- **tipo:** sistema/técnico;
- **descrição:** o sistema deve conseguir analisar a complexidade do código (por exemplo, a complexidade *Big-O* localmente no ambiente do usuário. A descrição dos algoritmos está na Seção 4.1.3;
- **pré-condições:** estrutura da AST;
- **pós-condições:** cota assintótica ou validação remota;

i) **analisar Complexidade no servidor por LLM:**

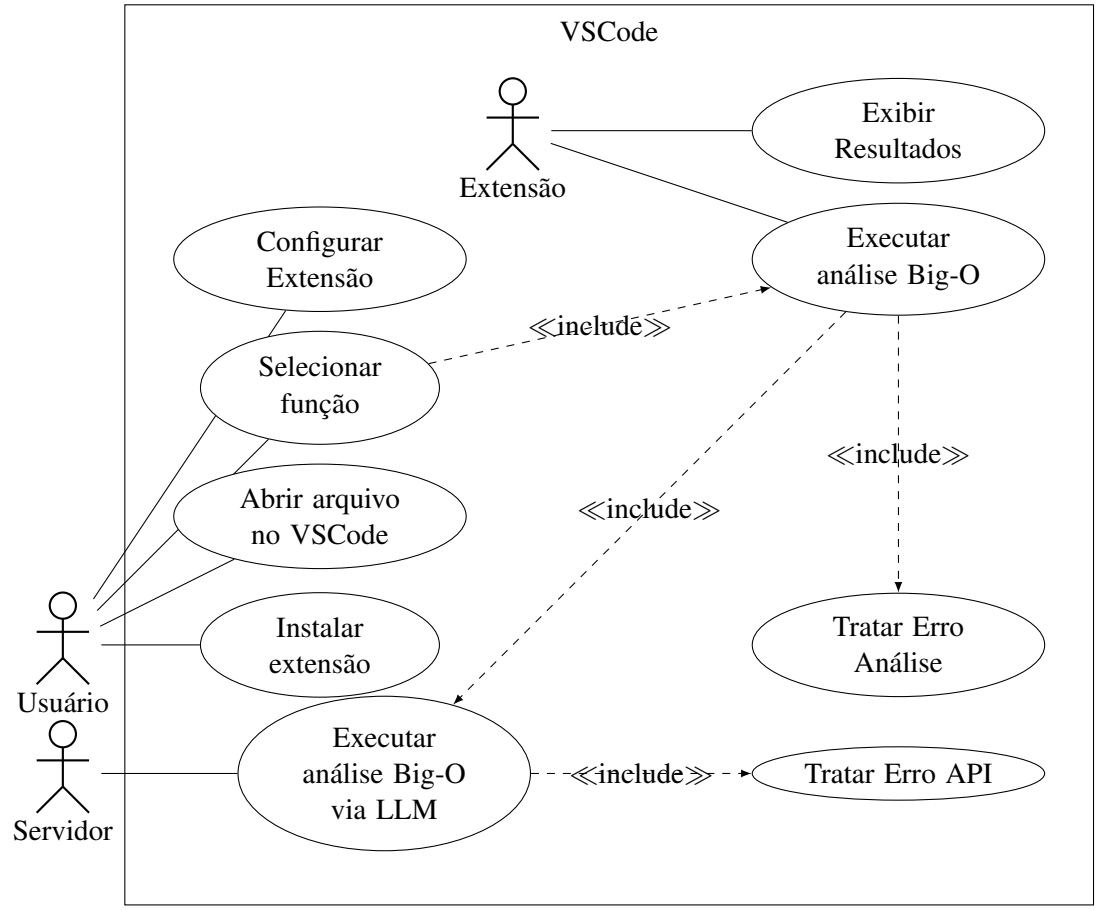
- **prioridade:** alta;
- **tipo:** sistema/técnico;
- **descrição:** o sistema deve conseguir analisar a complexidade do código no servidor usando o método LLM;

- **pré-condições:** chave de API e código;
- **pós-condições:** cota assintótica.

4.1.2 Casos de uso

Baseado nos requisitos, foram definidos os seguintes casos de uso identificados para a extensão e diagramados conforme a Figura 5:

Figura 5 – Diagrama de Casos de Uso.



Fonte: O Autor (2024)

a) **instalar extensão:**

- **ator:** usuário;
- **pré-condições:** o usuário tem o VSCode instalado;
- **fluxo principal:** o usuário seleciona a extensão na loja de extensões do VSCode e clica em instalar;
- **pós-condições:** a extensão está instalada e pronta para ser usada no VSCode;

b) **abrir arquivo no VSCode:**

- **ator:** usuário;
- **pré-condições:** o usuário tem um arquivo (código-fonte) que deseja abrir;

- **fluxo principal:** o usuário navega até o arquivo desejado e o abre no VSCode;
- **pós-condições:** o arquivo está aberto no VSCode. A extensão adiciona anotações para a análise de função acima de cada função. As anotações, textos e *links* que ficam atrelados à função e providenciam informações e ações relacionadas, exemplificado no *mockup* da Figura 6;

c) **selecionar função:**

- **ator:** usuário;
- **pré-condições:** o usuário tem um arquivo aberto no VSCode que contém funções anotadas;
- **fluxo principal:** o usuário navega até a função desejada no arquivo e seleciona a anotação;
- **pós-condições:** a função está selecionada e aciona gatilho para a análise;

d) **obter AST do código:**

- **ator:** usuário;
- **pré-condições:** o usuário selecionou uma função para a análise (gatilho acionado);
- **fluxo principal:** integrar com Clang[2] para obter a AST referente ao código selecionado (e às suas dependências);
- **pós-condições:** a análise *Big-O* é executada na AST selecionada e exibe o valor ao lado da anotação;

e) **executar análise *Big-O*:**

- **ator:** Extensão;
- **pré-condições:** gatilho de análise acionado e obtido AST do código;
- **fluxo principal:** efetua análise de *Big-O* da função;
- **pós-condições:** a análise *Big-O* é executada na função selecionada e exibe o valor ao lado da anotação;

f) **executar análise *Big-O* via LLM:**

- **ator:** servidor;
- **pré-condições:** a análise *Big-O* foi iniciada pelo usuário (gatilho acionado) e não conseguiu realizar localmente, solicitando-a ao servidor;
- **fluxo principal:** o servidor recebe a solicitação de análise e executa a análise *Big-O* na função;
- **pós-condições:** a análise *Big-O* é concluída e o resultado é enviado de volta para a extensão;

[2] CLANG. **Introduction to the Clang AST**. 2007. Disponível em: <https://clang.llvm.org/docs/IntroductionToTheClangAST.html>. Acesso em: 21 jun. 2024.

g) **tratar Erro Análise:**

- **ator:** extensão;
- **pré-condições:** ocorreu um erro durante a análise *Big-O*;
- **fluxo principal:** a extensão identifica o erro e exibe uma mensagem de erro para o usuário;
- **pós-condições:** o erro é tratado e o usuário é informado;

h) **tratar erro de API:**

- **ator:** servidor;
- **pré-condições:** ocorreu um erro durante a execução da análise *Big-O* via LLM;
- **fluxo principal:** o servidor identifica o erro e envia uma mensagem de erro para a extensão;
- **pós-condições:** o erro é tratado e a extensão é informada;

i) **exibir resultados:**

- **ator:** extensão;
- **pré-condições:** a análise da função foi concluída;
- **fluxo principal:** a extensão exibe os resultados da análise para o usuário junto à anotação da função analisada;
- **pós-condições:** vazio;

j) **configurar extensão:**

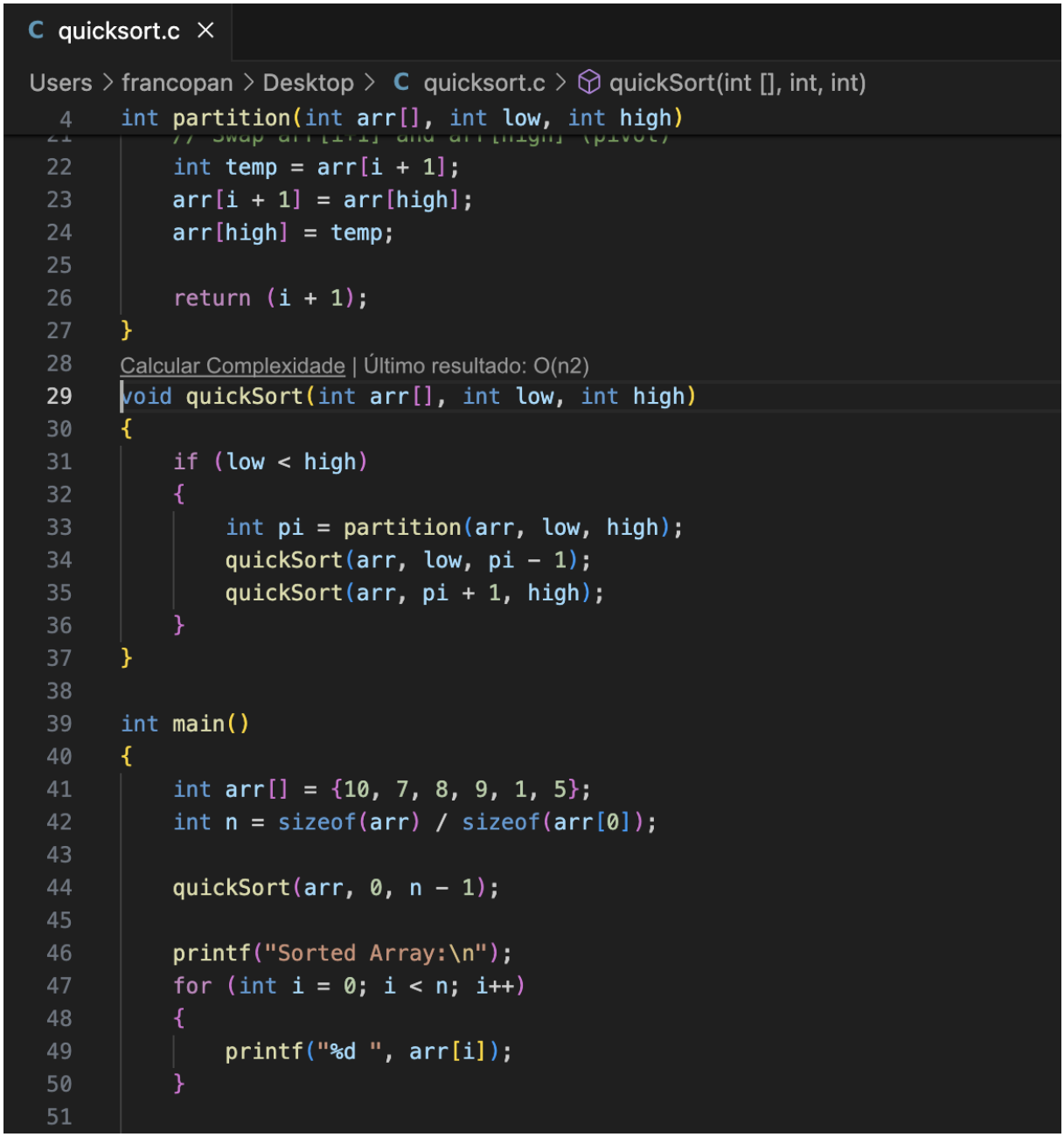
- **ator:** usuário;
- **pré-condições:** extensão está instalada;
- **fluxo Principal:** o usuário acessa a página de configuração da extensão e pode determinar qual o tempo limite para a análise;
- **pós-condições:** *feedback* visual de confirmação de que a configuração foi salva com sucesso. Este feedback é normalmente providenciado automaticamente pelo Visual Studio Code, evidenciado pela remoção de um ícone de "bolinha" (indicador de modificação) ao lado do nome do arquivo na aba do editor. Outra indicação, o nome do arquivo, que inicialmente aparece em itálico (indicando que o arquivo foi modificado), altera-se para o formato normal (não itálico), indicando que as alterações foram salvas.

Para aprofundar em relação aos requisitos necessários, foram levantados algoritmos, desenhos de arquitetura e outras informações que auxiliarão no desenvolvimento.

4.1.3 Algoritmos para a análise de complexidade

Com o intuito de facilitar a implementação, foram utilizados como base (mas não se limitando a) os pseudocódigos abaixo. Com cada nó identificado e categorizado, pode-se atribuir

Figura 6 – *Mockup* de tela com anotação de código



Fonte: O Autor (2024)

um cálculo para cada um deles, conforme Algoritmo 3. Implementações ausentes eventualmente de um pseudocódigo seguirão a mesma linha de implementação.

Algoritmo 3 – Pseudocódigo para base da análise de complexidade

```
1 funcao analisarComplexidadeAST(ast)
2     retorne analisarNo(ast)
3
4 funcao analisarNo(no)
5     se no == laco
6         retorne analisarLaco(no)
```

```
7       senao se no == condicional
8           retorne analisarCondicional(no)
9       senao se no == composicao
10          retorne analisarComposicao(no)
11      senao se no == chamada de funcao
12          retorne analisarChamadaFuncao(no)
13      senao
14          retorne 1
```

Se um nó da AST possui filhos, significa que ele é um nó de composição e precisa saber a complexidade desses componentes na totalidade, conforme o Algoritmo 4.

Algoritmo 4 – Pseudocódigo para a análise de complexidade de composição

```
1  funcao analisarComposicao(no)
2      complexidade_composicao = 0
3      para cada componente em no.componentes faca
4          complexidade_composicao += analisarNo(componente)
5      retorne complexidade_composicao
```

Outras estruturas mais complexas, como condicionais, laços e chamadas de funções, são descritas nos Algoritmos 5, 6 e 7.

Algoritmo 5 – Pseudocódigo para a análise de complexidade de condicionais

```
1  funcao analisarCondicional(no)
2      complexidade_condicional = analisarNo(no.condicao)
3      complexidade_verdadeiro = analisarNo(no.blocoVerdadeiro)
4      complexidade_falso = existe no.blocoFalso ?
5          analisarNo(no.blocoFalso) : 1
6      retorne max(complexidade_verdadeiro, complexidade_falso)
7          + complexidade_condicional
```

A dificuldade em relação à análise de laços se refere, principalmente, sobre o crescimento da complexidade quando as variáveis de controle dos laços são dependentes entre si, fazendo com que o laço execute mais ou menos vezes. Já para laços do tipo *while* existe a dificuldade de determinar um momento para a parada, precisando de certa maneira executar o laço ou simular sua execução para ter uma estimativa, tornando a análise estática inviável.

Algoritmo 6 – Pseudocódigo para a análise de complexidade de laços

```
1  funcao analisarLaco(no)
2      complexidade_iteracao = obterComplexidadeIteracao(no)
3      complexidade_corpo = 1
4      para cada filho de no faca
5          complexidade_corpo *= analisarNo(filho)
6      retorne complexidade_iteracao * complexidade_corpo
7
8  funcao obterComplexidadeIteracao(no)
```

```
    se no.tipoLaco == 'FOR'
        inicio = avaliarExpressao(no.inicio)
        fim = avaliarExpressao(no.fim)
        incremento = avaliarExpressao(no.incremento)
        se incremento == 0
            retorne 'Erro: Incremento nao pode ser zero'
        senao
            retorne (fim - inicio) / incremento + 1
    senao se no.tipoLaco == 'WHILE' ou no.tipoLaco == 'DO_WHILE'
        retorne avaliarComplexidadeWhile(no) // TODO
    senao
        retorne 'Tipo de laco nao reconhecido'
```

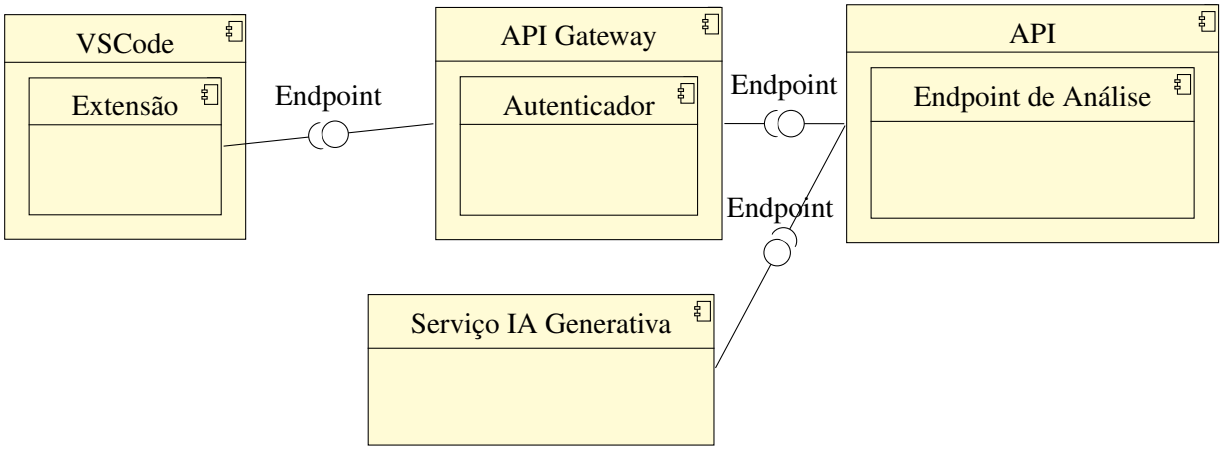
Algoritmo 7 – Pseudocódigo para a análise de complexidade de chamadas de função

```
funcao analisarChamadaFuncao(no)
    retorne analisarNo(no.funcao) +
        sum(analisarNo(arg) para arg em no.argumentos)
```

4.1.4 Arquitetura

Para exemplificar a interação entre os componentes deste sistema, estão diagramadas na Figura 7 as conexões realizadas entre os componentes.

Figura 7 – Interação entre componentes



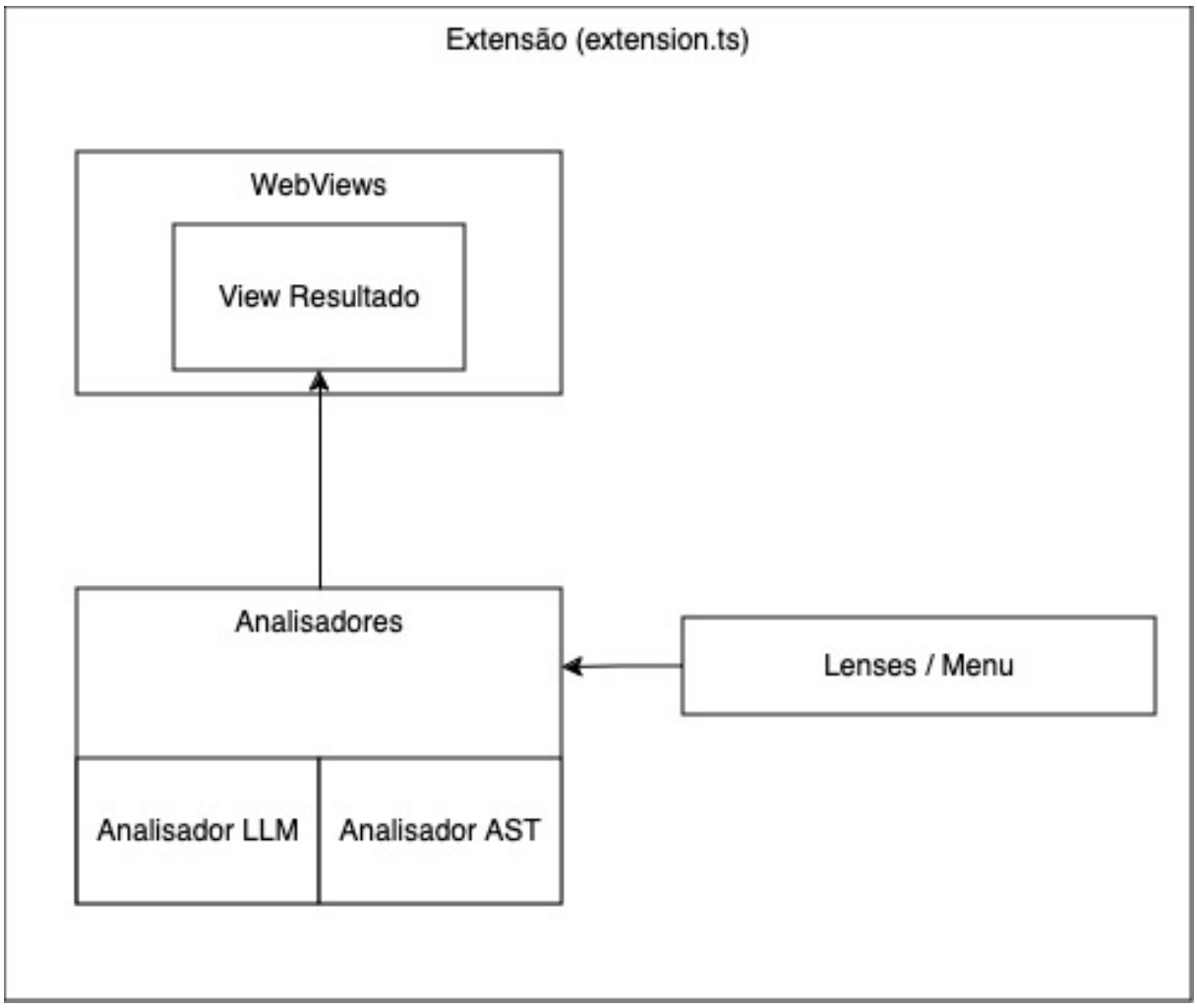
Fonte: O Autor (2024)

A extensão interage com o código-fonte do usuário, analisando funções e estruturas de controle para calcular sua complexidade de tempo, sendo então exibida diretamente no editor.

O desenvolvimento da extensão foi realizado conforme SDK (Seção 4.1.4.1), providenciado pela Microsoft. A linguagem de programação da extensão é *Typescript*, uma vez que é (junto ao *Javascript*) uma das linguagens suportadas.

A arquitetura da extensão, conforme Figura 8, pode ser separada por módulos: a criação de uma classe *analisadora*; a exibição das informações via um painel *web*; menus e links de ações para a interação com o usuário. Cada uma dessas partes foi separada em um diretório a parte, conforme Figura 9, a fim de separar as preocupações do código, trabalhando de maneira independente.

Figura 8 – Diagrama da arquitetura da extensão.



Fonte: O Autor (2024)

4.1.4.1 Biblioteca para desenvolvimento

A Microsoft disponibiliza um guia online e gratuito para desenvolvimento de extensões para o VS Code[3] no qual consta uma seção para informar como desenvolver ferramentas que interagem com o código-fonte.

[3] MICROSOFT. **Your First Extension | Visual Studio Code Extension API**. Microsoft, 2024. Disponível em: <https://code.visualstudio.com/api/get-started/your-first-extension>. Acesso em: 27 fev. 2024.

4.1.4.2 Servidor de protocolo de linguagem

Uma das dificuldades de se interpretar um determinado código é que cada linguagem de programação possui uma definição própria e, portanto, fazer o *parsing* de cada linguagem pode ser exaustivo. Felizmente, existe um protocolo criado pela própria Microsoft que auxilia na padronização da comunicação do editor e do analisador, o *Language Server Protocol (LSP)* (MICROSOFT, 2024a). O LSP permite que editores consigam se comunicar com um servidor de linguagem mediante um cliente. Em outras palavras, não se torna necessário reimplementar o analisador para cada editor de texto, desde que eles implementem o protocolo. Portanto, conforme a Figura 10, é possível evidenciar que isso simplifica o desenvolvimento, uma vez que com apenas uma implementação de um servidor, basta implementar os clientes para cada editor.

4.1.4.3 Anotação de código/*CodeLens*

Uma das maneiras de proporcionar informações relevantes ao usuário desenvolvedor no VS Code é uma API chamada de CodeLens (ANDERSON, 2017), que permite anotar o código e realizar ações por um clique. Isso possibilita que sejam identificadas funções e, para cada função, delegar a análise do código perante a requisição do usuário, tornando a utilização de recursos mais balanceada.

> Por exemplo, para validar corretamente um arquivo, o *Language Server* precisa analisar uma abundância de arquivos, construir árvores de sintaxe abstratas para eles e realizar análises estáticas do programa. Essas operações podem incorrer em uso significativo de Central Processing Unit (CPU) e memória e precisam garantir que o desempenho do VS Code permaneça inalterado (MICROSOFT, 2024a).

4.2 IMPLEMENTAÇÃO

As tecnologias principais utilizadas na implementação incluem o `TypeScript` e a API do *Visual Studio Code*, que permite a integração com o editor de código. A extensão consegue identificar algoritmos comuns e estimar sua complexidade temporal com base em uma análise de uma árvore sintática abstrata do código-fonte. A extensão foi desenvolvida utilizando os seguintes módulos:

4.2.1 Implementação de uma Extensão

Para a implementação de uma extensão para o Visual Studio Code, utilizou-se a ferramenta *Yeoman*(MICROSOFT, 2024d), que facilita a criação de um projeto inicial com a estrutura básica, também conhecida como código *boilerplate*. Através dessa ferramenta, é possível responder a uma série de perguntas sobre as preferências do projeto, como a linguagem de programação, nome, repositório, sistema de *build*, entre outros parâmetros. Dessa forma, o desenvolvedor obtém um ponto de partida eficiente para o desenvolvimento da extensão.

A instalação do *Yeoman* pode ser realizada por meio do Node Package Manager (NPM), utilizando o comando:

```
npm install --global yo generator-code
```

Após a instalação, a ferramenta pode ser utilizada para gerar o esqueleto da extensão com o comando:

```
yo code
```

Uma vez gerado o projeto, o desenvolvimento da extensão é iniciado a partir do arquivo *extension.ts* ou *extension.js*, que contém o código inicial. A principal função desse arquivo é a função *activate*, que é executada sempre que a extensão é instalada ou ativada no Visual Studio Code. Essa função é executada na inicialização do editor, tornando-se o ponto de partida para qualquer lógica adicional da extensão.

O projeto gerado pelo *Yeoman* segue uma estrutura padrão que facilita o desenvolvimento. O arquivo *package.json* contém informações sobre a extensão, como nome, descrição, versão, dependências e scripts. A pasta *src* contém o código fonte, enquanto a pasta *out* é gerada após a compilação do código TypeScript (caso o projeto use TypeScript).

4.2.1.1 Codificação

A partir daí, o projeto pode ser desenvolvido de forma flexível, já que a extensão é construída utilizando *JavaScript* ou *TypeScript*, linguagens que permitem adotar os mesmos padrões de projeto encontrados em aplicações *Node.js*. Isso ocorre porque as extensões para o Visual Studio Code rodam em um ambiente baseado no *Node*, proporcionando compatibilidade com os padrões comuns dessa plataforma.

Para a integração com as APIs do Visual Studio Code, basta importar a biblioteca `vscode` no código da extensão, utilizando a seguinte instrução:

```
import * as vscode from 'vscode';
```

Após a importação, as funcionalidades do editor podem ser acessadas conforme as necessidades da extensão, seguindo as diretrizes e a documentação oficial da Microsoft. Exemplos de funcionalidades comuns incluem o uso de *settings*, *webviews*, *code lens* e outras APIs oferecidas pela plataforma (MICROSOFT, 2024d).

A seguir, um exemplo de uma extensão simples que registra um comando que exibe uma mensagem quando acionado:

```
1 import * as vscode from 'vscode';
2
3 export function activate(context: vscode.ExtensionContext) {
4     let disposable = vscode.commands.registerCommand('extension.helloWorld',
```

```
5
6       () => {
7           vscode.window.showInformationMessage('Hello, World!');
8       });
9
10      context.subscriptions.push(disposable);
11  }
```

Nesse exemplo, ao executar o comando `extension.helloWorld` no editor, uma mensagem simples será exibida.

Para adicionar pacotes externos, como bibliotecas que possam ser necessárias para a extensão, basta usar o comando `npm install <package-name>`.

4.2.1.2 Testes

É essencial garantir que a extensão funcione corretamente. Para isso, pode-se utilizar frameworks de teste como `Mocha` ou `Jest`. O *Visual Studio Code* oferece suporte nativo para testes unitários, permitindo que os desenvolvedores integrem seus testes diretamente no fluxo de desenvolvimento da extensão (MICROSOFT, 2024d).

4.2.1.3 Publicação

Por fim, para a publicação da extensão, utiliza-se a ferramenta `VSCE (@vscode/vsce)` (MICROSOFT, 2024c), que permite empacotar e publicar a extensão na loja oficial do Visual Studio Code. O processo de publicação envolve dois comandos principais:

```
1 vsce package
```

O comando acima cria o arquivo *vsix* da extensão, que contém todos os arquivos necessários para a instalação. Em seguida, para publicar a extensão na loja da Microsoft, basta executar:

```
1 vsce publish
```

Além disso, é possível publicar a extensão em outras lojas de pacotes ao realizar o upload do arquivo *vsix* gerado no processo de *build*.

4.2.2 Módulo de Análise de Algoritmos

Este módulo é responsável por realizar a análise do código-fonte. Ele identifica os principais algoritmos de ordenação, busca e outros tipos de operações algorítmicas, como recursão e laços aninhados. A análise é feita por meio de técnicas de análise estática do código, identificando padrões que correspondem a comportamentos algorítmicos conhecidos.

O desenvolvimento de um analisador de complexidade algorítmica baseado em Modelos de Linguagem (LLM) visa aproveitar o poder de redes neurais treinadas para compreender e gerar análises de código-fonte. Em vez de depender de uma AST gerada por um compilador, como na abordagem anterior, o LLM se comunica diretamente com a API de um modelo de linguagem para fornecer uma análise mais semântica e contextual do código.

4.2.2.1 Estrutura do Analisador LLM

A classe `LLMAnalyzer` foi implementada para realizar análises de complexidade de código-fonte usando um modelo de linguagem baseado em IA. A interação com o modelo de linguagem ocorre por meio de uma API, que recebe o código como entrada e retorna uma análise textual que descreve a complexidade do código fornecido.

O código-fonte da classe `LLMAnalyzer` está estruturado da seguinte forma:

- **Parâmetros de Configuração**: O construtor da classe recebe dois parâmetros principais:
  - `llmAPIURL`: A Uniform Resource Locator (URL) da API que fornece acesso ao modelo de linguagem. O padrão pode ser configurado para utilizar diferentes endpoints de modelos de linguagem.
  - `model`: O nome do modelo de linguagem utilizado, que por padrão é `mistral`, mas pode ser alterado para outros modelos como `gpt-3.5` ou `gpt-4` conforme necessário.
- **Análise de Código**: A função `analyze` recebe um código em formato de *string* e o envia para a API do modelo de linguagem. A entrada é fornecida em forma de um `prompt`, conforme Algoritmo 8, no qual o código é inserido dinamicamente. O modelo de linguagem, então, processa o código e retorna uma análise detalhada.
- **Resposta do Modelo**: A resposta do modelo é um texto que descreve a complexidade do código, ou um erro, dependendo da qualidade do código de entrada. A resposta é então processada e convertida em um objeto *AnalysisResult*.
- **Análise da Resposta**: A função *parseAnalysisResult* é responsável por processar a resposta textual do modelo, sendo inicialmente recebida como uma string. O texto é esperado em formato JSON, permitindo uma conversão direta para um objeto *AnalysisResult*.

Algoritmo 8 – *Prompt*

```
1 export const prompts = {
2     analyzeCode: `Code: \`\`\`c\n{{code}}\n\`\`\`
3         Analyze the code for its time complexity in Big O notation.
4         IN ORDER FOR ME TO PARSE THE RESULT, RETURN A JSON OBJECT WITH
5         KEYS "bigO" AND "message". USE "message" FOR ANY EXPLANATION OR
```

```
6           ERROR, BUT DO NOT INCLUDE ANY OTHER TEXT BEFORE OR AFTER
7           THE JSON. IGNORE FUNCTION NAMES AND COMMENTS.
8
9           Expected result format:
10
11          {
12              "bigO": "O(log(n) * log(n))",
13              "message": "Explanation of the time complexity"
14          }
15
16          OR
17
18          {
19              "bigO": "O(2^n)",
20              "message": "Explanation of the time complexity"
21          }
22
23          OR
24
25          {
26              "bigO": null,
27              "message": "Explanation of why cannot give complexity"
28          }`,
29  };
```

4.2.2.1.1 Fluxo de Trabalho

O fluxo de trabalho da análise de código com o *LLMAnalyzer* é o seguinte:

1. O código-fonte a ser analisado é passado como parâmetro para a função `analyze`.
2. A função `analyze` cria um `prompt` dinâmico, onde o código é inserido no formato pré-definido.
3. O `prompt` é enviado para a API do modelo de linguagem utilizando uma solicitação *POST* via `axios`.
4. A resposta da API é recebida como uma string, que contém uma análise do código gerada pelo modelo de linguagem.
5. A resposta é processada para extrair a análise de complexidade, convertendo o texto para o formato `AnalysisResult`.
6. A análise final é retornada como resultado.

O código de exemplo da classe `LLMAnalyzer` em JavaScript é mostrado abaixo:

Algoritmo 9 – Código de análise usando LLM

```
import axios from "axios";
import { prompts } from "../constants/prompts.const";
import { AnalysisResult } from "../models/analysis-result.model";
import { Analyzer } from "./analyzer.interface";

export class LLMAnalyzer implements Analyzer {
    private readonly llmAPIURL: string;
    private readonly model: string;

    constructor(llmAPIURL: string =
            'http://127.0.0.1:11434/api/generate',
            model: string = 'mistral') {
        this.llmAPIURL = llmAPIURL;
        this.model = model;
    }

    public async analyze(code: string): Promise<AnalysisResult> {
        const prompt = prompts.analyzeCode.replace('{{code}}', code);
        try {
            const response = await axios.post(this.llmAPIURL, {
                model: this.model,
                prompt: prompt,
                stream: false
            });

            const responseText = response.data.response.trim();
            return this.parseAnalysisResult(responseText);
        } catch (error) {
            console.error('Error while analyzing code with LLM:',error);
            throw new Error('Failed to analyze with LLM');
        }
    }

    private parseAnalysisResult(responseText: string): AnalysisResult {
        try {
            return JSON.parse(responseText) as AnalysisResult;
        } catch (error) {
            console.error('Failed to parse response:',responseText);
            throw new Error('Invalid JSON response');
        }
    }
}
```

4.2.2.1.2 Considerações Técnicas

- **Uso de Modelos de Linguagem**: A principal vantagem do uso de modelos de linguagem (LLMs) para análise de complexidade é a flexibilidade e a capacidade de entender contextos mais amplos do código, além de conseguir fornecer explicações detalhadas em linguagem natural. Isso permite uma análise mais rica e acessível do que a simples avaliação de estruturas de código baseadas em regras fixas.
- **Dependência de API Externa**: O modelo de linguagem é acessado por meio de uma API externa, o que significa que a análise depende da disponibilidade e desempenho da API. Além disso, a precisão e a qualidade da análise variam dependendo do modelo utilizado e da clareza do código de entrada.
- **JavaScript Object Notation (JSON) como Formato de Resposta**: A resposta do modelo é convertida para um formato JSON, permitindo que as informações de complexidade sejam extraídas de maneira estruturada. A análise do código é feita de forma semântica, em vez de seguir uma análise sintática detalhada como em abordagens tradicionais baseadas em AST.

4.2.2.2 Desenvolvimento de um Analisador usando AST

O desenvolvimento de um analisador de complexidade algorítmica utilizando a AST visa fornecer uma análise automatizada da complexidade de tempo de códigos em linguagens como C e outras suportadas pelo compilador *Clang*. O processo envolve a análise do código-fonte para identificar padrões estruturais como laços, condicionais, chamadas de funções e operadores binários, a fim de calcular sua complexidade temporal.

4.2.2.2.1 Estrutura de Análise e Ferramentas Utilizadas

A análise de complexidade começa com a captura da AST de um código-fonte. A ferramenta desenvolvida utiliza o Clang, uma poderosa infraestrutura de compilação que gera AST no formato JSON a partir de código C pelo comando `clang -Xclang -ast-dump=json`. Este comando é executado programaticamente a partir da IDE, Visual Studio Code, usando a API `exec` do *Node.js*, que permite a execução de comandos do sistema operacional de maneira assíncrona.

A estrutura da AST gerada pelo *Clang* é organizada hierarquicamente, onde cada nodo representa uma construção do código, como uma declaração de variável, um laço *for*, uma instrução condicional *if*, entre outros. A partir dessa árvore, o analisador pode percorrer e avaliar a complexidade dos diferentes elementos, como detalhado abaixo.

4.2.2.2.2 Análise de Complexidade: Algoritmos e Estratégias

O código implementa um analisador de complexidade que segue os seguintes princípios fundamentais:

- **Percurso da AST**: A função `traverse` percorre a árvore de sintaxe abstrata, analisando recursivamente cada nodo. O analisador avalia a complexidade de cada nó e soma os resultados em uma contagem total de operações.
- **Identificação de Padrões**: O analisador identifica diferentes tipos de estruturas de controle e operações:
  - **Laços (`ForStmt`, `WhileStmt`)**: A complexidade de um laço é determinada pelo número de iterações. Para os laços *for*, o analisador calcula a quantidade de iterações baseada na condição de término e no incremento. Já para os laços *while*, a condição de continuidade é avaliada para determinar o número de execuções.
  - **Condicionais (`IfStmt`)**: Para instruções condicionais, o analisador calcula a complexidade de ambos os ramos (verdadeiro e falso) e escolhe o ramo com a maior complexidade, já que ambos os caminhos podem ser executados dependendo da condição.
  - **Operadores binários (`BinaryOperator`)**: Operações como soma, subtração, multiplicação e divisão são consideradas para avaliar o custo de execução, sendo que operações de maior complexidade (como multiplicações e divisões) aumentam a contagem de operações.
- **Avaliação de Expressões e Funções**: O analisador também consegue avaliar expressões em condições e operadores binários, além de calcular o número de operações realizadas em chamadas de funções, como na análise de expressões do tipo `CallExpr` e `BinaryOperator`.
- **Tabela de Variáveis**: Durante a análise, é mantida uma tabela de variáveis que armazena os valores de variáveis locais, permitindo a avaliação precisa de expressões e incrementos.

4.2.2.2.3 Cálculo da Complexidade Assintótica (Big O)

O cálculo da complexidade assintótica do código é feito com base na contagem de operações coletadas durante a análise. A função `categorizeGrowth` recebe um array de contagens de operações para diferentes tamanhos de entrada `n` e calcula a taxa de crescimento das operações conforme `n` aumenta. A partir dessa taxa de crescimento, o código classifica a complexidade em uma das seguintes categorias, seguindo como base o método de fator de classificação utilizado por Costa *et al.* (2014) no projeto IGED.

- $O(1)$: complexidade constante.
- $O(\log n)$: complexidade logarítmica.
- $O(n)$: complexidade linear.
- $O(n \log n)$: complexidade linearítmica.
- $O(n^2)$: complexidade quadrática.
- $O(n^3)$: complexidade cúbica.
- $O(2^n)$: complexidade exponencial.
- $O(n!)$: complexidade fatorial.
- $O(n^k)$: complexidade polinomial ou superior.

4.2.3 Restrições

Para realizar a implementação e a validação no cronograma estabelecido, foi necessário definir algumas restrições referentes à implementação:

a) **análise de funções recursivas**: não foi realizada a análise de funções recursivas, visto que, em relação ao material encontrado, é um cenário mais complexo e, algumas vezes, com necessidade de contexto, o que poderia inviabilizar a entrega do projeto;

b) **limite de tempo para validação**: conforme abordado previamente na Seção 2.4, não é possível saber quando um programa para de executar, sendo necessário ter um limite de tempo máximo estabelecido;

c) **linguagem de programação analisada**: foi analisada a linguagem de programação C em sua versão estável mais recente. Não foi suportada outra linguagem no momento, uma vez que se faz necessária a AST e ela é dependente do LSP implementado;

d) **paralelismo**: não foram considerados casos de programação paralela, as operações foram consideradas executadas em sequência uma após a outra;

e) **subconjunto da linguagem**: foi analisado um subconjunto da linguagem C contendo os elementos: chamadas de funções; inteiros; pontos-flutuantes; caracteres; vetores; laços *for*, *while* e *do-while*; condicionais simples e completas (*if, else if, else* e atribuições. Os demais componentes da linguagem foram descartados da análise e considerados O(1);

f) **compilador**: foi utilizado o compilador *Clang* como base para esse trabalho, portanto, a extensão não irá suportar outros compiladores.

4.2.4 Interface com o Usuário

A interface com o usuário foi projetada para exibir o resultado da análise de *Big O* diretamente no editor de código do *VSCode*. O resultado sempre é aberto em uma janela denominada *webview*, conforme Algoritmo 10, pois ela apresenta o conteúdo em formato similar ao de navegadores, conforme Algoritmo 11 e demonstrado na Figura 11. O usuário recebe um painel com o resultado ou uma notificação com alguma mensagem apropriada.

Para realizar a chamada da *webview*, foi criada uma classe *singleton* em que se instancia a *webview* e *renderiza* o *template*.

Algoritmo 10 – Código para *Template* de *Webview*

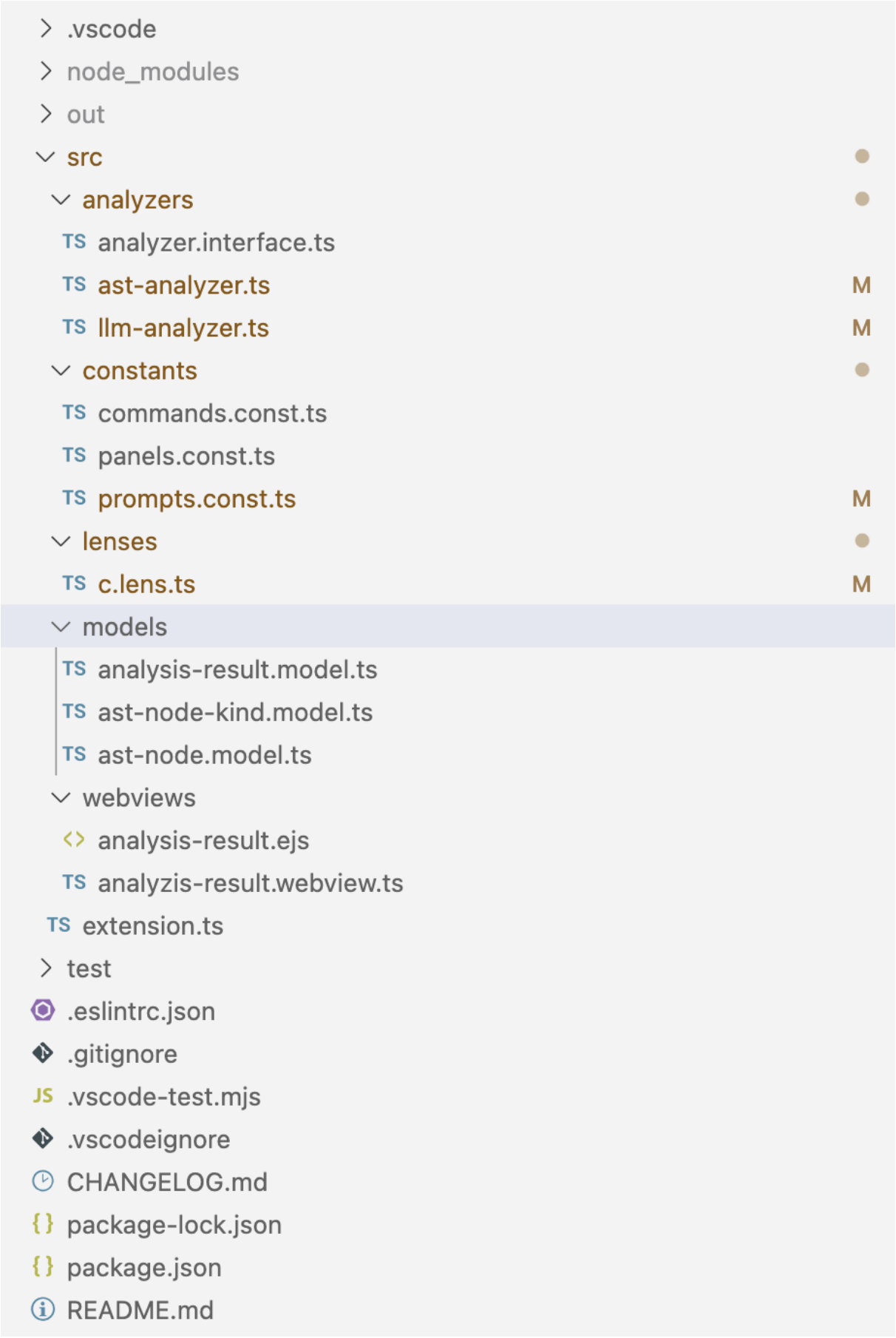
```
1
2  export class AnalysisResultWebview implements vscode.Disposable {
3
4      private isDebugMode: boolean = false;
5      private extensionPath: string = '';
6      private panel?: vscode.WebviewPanel;
7      private static instance: AnalysisResultWebview;
8
9      public static getInstance(): AnalysisResultWebview {
10         if (!AnalysisResultWebview.instance) {
11             AnalysisResultWebview.instance = new AnalysisResultWebview();
12         }
13         return AnalysisResultWebview.instance;
14     }
15
16     public async showAnalysisResult(result: AnalysisResult) {
17         this.panel?.dispose();
18         this.panel = vscode.window.createWebviewPanel(
19             panels.analysisResult,
20             'Analysis Result',
21             vscode.ViewColumn.Beside,
22             {}
23         );
24         this.panel.webview.html =
25             this.renderTemplate('analysis-result.ejs', result)
26                 ?? 'Error loading result';
27     }
28
29     public dispose() {
30         this.panel?.dispose();
31     }
32
33     public setIsDebugMode(value: boolean): void {
34         this.isDebugMode = value;
35     }
```

```
36
37      public setExtensionPath(path: string): void {
38          this.extensionPath = path;
39      }
40
41      private renderTemplate(fileName: string, data: ejs.Data): string {
42          const templateDir = this.isDebugMode ? 'src/webviews' : '';
43          const templatePath = join(this.extensionPath,
44              templateDir,
45              fileName);
46          const template = readFileSync(templatePath, 'utf8');
47          const htmlContent = render(template, data);
48
49          return htmlContent;
50      }
51
52
53  }
```

Algoritmo 11 – *Template* HTML

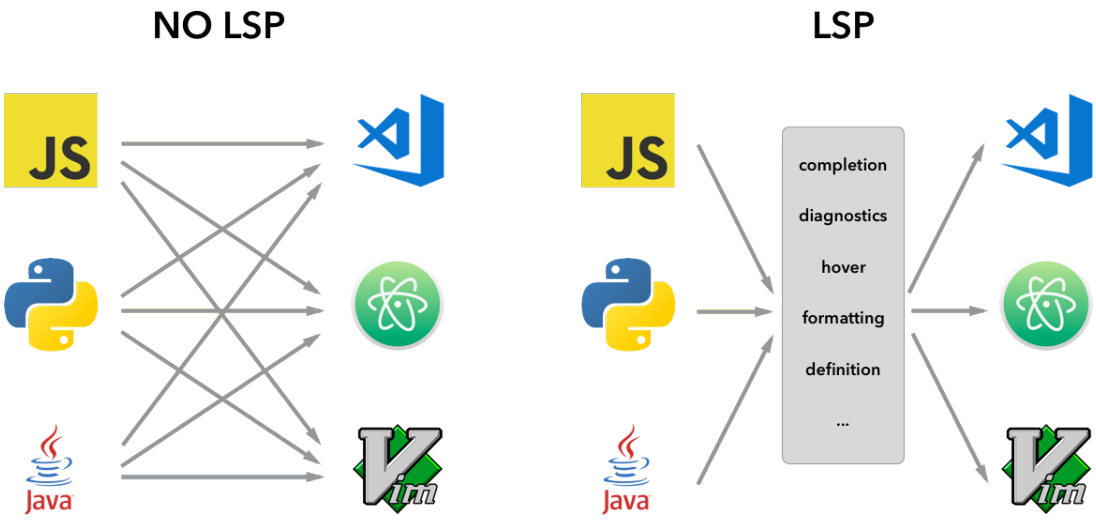
```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Analysis Result</title>
6  </head>
7  <body>
8      <h1>Analysis Result</h1>
9      <p><strong>Big O:</strong> <%= bigO || 'N/A' %></p>
10     <p><strong>Message:</strong> <%= message %></p>
11 </body>
12 </html>
```

Figura 9 – Árvore de arquivos do projeto.



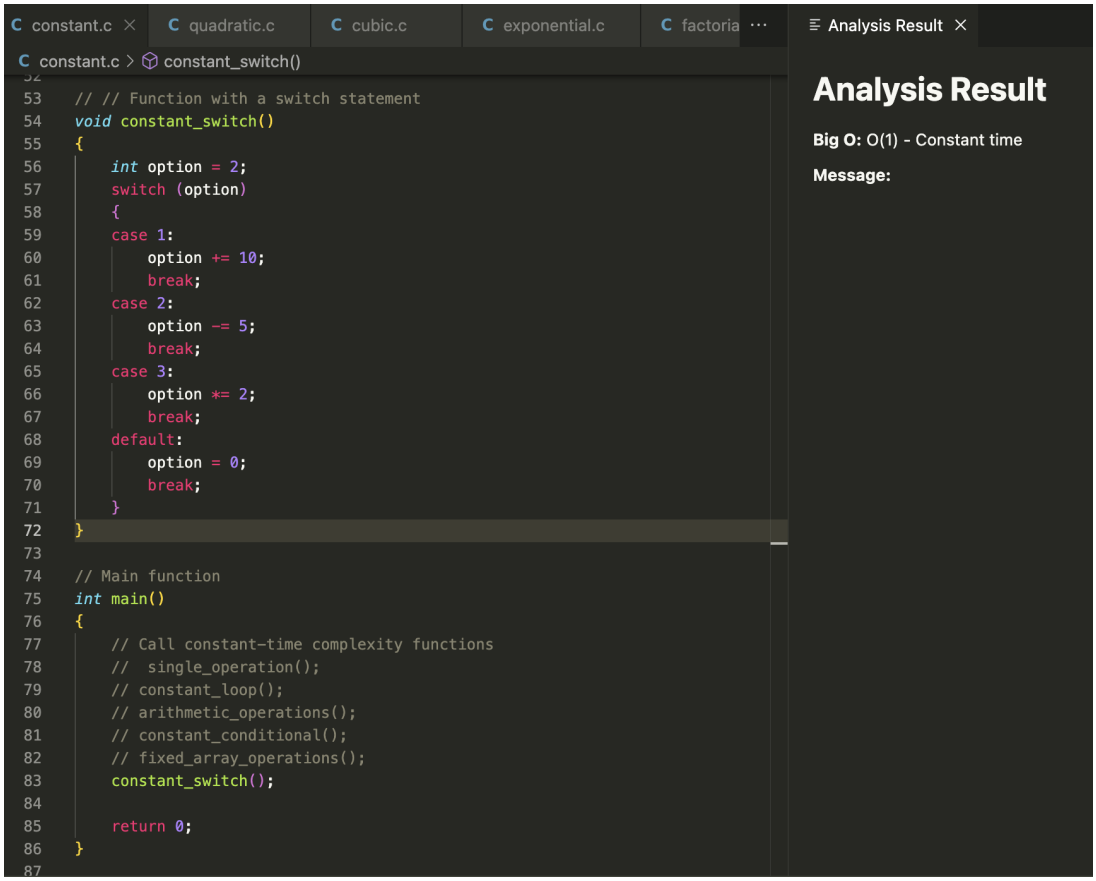
Fonte: O Autor (2024)

Figura 10 – Comparação entre utilização ou não de um servidor de linguagem



Fonte: Microsoft (2024)

Figura 11 – Tela com resultado da análise



Fonte: O Autor (2024)

5 TESTES E VALIDAÇÃO DA EXTENSÃO

A ferramenta desenvolvida foi projetada para avaliar a complexidade assintótica de funções utilizando dois métodos principais: análise da AST e um modelo baseado em LLM. Os resultados dos testes conduzidos são apresentados no apêndice A. Esta seção discute os resultados, identificando acertos, erros e possíveis melhorias.

5.1 RESULTADOS GERAIS

Os testes envolveram funções com complexidades variadas, desde constantes ($O(1)$), exponenciais ($O(2^n)$) e logarítmicas ($O(\log n)$). A precisão do sistema foi avaliada comparando o resultado esperado para cada função com os valores obtidos pela análise da AST e pelo LLM.

- **Desempenho da AST**: A análise da AST demonstrou resultados mistos. O percentual de acertos foi baixo (cerca de 34,37%), e, embora o método tenha sido preciso para identificar complexidades lineares ($O(n)$), quadráticas ($O(n^2)$), cúbicas ($O(n^3)$) e exponenciais em laços simples, falhou ao lidar com estruturas mais complexas como somatórios acumulativos (e.g., *linear_cumulative_sum*) e abordagens dinâmicas (e.g., *factorial_dynamic*). Essas falhas indicam limitações na interpretação de padrões não triviais na estrutura do código.
- **Desempenho do LLM**: O LLM obteve uma precisão consideravelmente superior, acertando 96,87% dos casos, identificando corretamente complexidades para quase todas as funções. Em particular, ele se destacou nas análises logarítmicas e em cenários com múltiplos laços aninhados. Entretanto, erros ocorreram em funções simples como *constant_loop*, onde o LLM inferiu incorretamente uma complexidade linear ($O(n)$).

5.2 CASOS ESPECÍFICOS

- **Análise de Loops Lineares e Condicionais**: Enquanto a AST determinou incorretamente a complexidade como exponencial em funções como *linear_with_condition* e *linear_independent_loops*, o LLM conseguiu identificar corretamente a linearidade das operações. Esse resultado evidencia a capacidade do LLM de compreender padrões de controle condicionais.
- **Operações Quadráticas e Cúbicas**: Funções com complexidades quadráticas e cúbicas foram corretamente analisadas em sua maioria. A AST, no entanto, apresentou dificuldades em funções com múltiplos padrões, como *quadratic_array_operations* e *cubic_array_operations*, sugerindo uma combinação incorreta de resultados.

- **Complexidades Logarítmicas**: Funções como *logarithmic_loop* e *binary_search* evidenciaram as limitações da AST, que não conseguiu identificar padrões logarítmicos, enquanto LLM obteve resultados precisos.
- **Funções Exponenciais e Fatoriais**: O LLM demonstrou alta precisão em determinar complexidades exponenciais. Entretanto, a AST falha consistentemente nesses cenários, sendo necessárias melhorias no reconhecimento de chamadas recursivas, conforme apontada limitação no planejamento desse projeto.

5.3 COMPARAÇÕES ENTRE TIPOS DE ANÁLISE

- A flexibilidade dos modelos de linguagem por Inteligência Artificial (IA) permite uma análise mais adaptável, capaz de lidar com uma ampla gama de padrões de código. O modelo pode fornecer explicações detalhadas e acessíveis sobre a complexidade do código, facilitando o entendimento por desenvolvedores não especializados em análise de desempenho.
- A precisão da análise por IA depende da qualidade do modelo de linguagem, que pode não ser sempre confiável para todos os tipos de código. A análise depende mais da entrada de dados e do contexto do código, o que pode resultar em variações na qualidade da resposta.
- A análise utilizando AST permite uma resposta mais rápida e sem a necessidade de um servidor web e um modelo de LLM rodando, no entanto, a variação e complexidade é tamanha, que cobrir a maioria dos casos torna-se uma tarefa muito complexa.

5.4 CONCLUSÃO DOS TESTES

Os resultados dos testes indicam que o uso combinado de AST e LLM pode fornecer análises complementares, onde o LLM compensa as limitações da análise estática. A principal limitação da AST é sua incapacidade de lidar com padrões avançados, como laços condicionais, recursão e operações acumulativas. Por outro lado, o LLM, apesar de ser mais preciso, apresenta falhas em cenários extremamente simples, indicando uma potencial "super análise" de padrões triviais. Esses resultados sugerem que futuras melhorias devem focar em:

- Refinar os algoritmos da AST para capturar padrões mais complexos;
- Otimizar o LLM para evitar inferências incorretas em funções simples;
- Integrar os resultados de ambas as abordagens para uma análise mais robusta.

6 CONSIDERAÇÕES FINAIS

A análise assintótica de complexidade de algoritmos representa um desafio significativo na computação. A teoria subjacente é robusta, mas a automatização desse processo ainda é uma área emergente, oferecendo um vasto campo para exploração. Esta análise não só otimiza o desenvolvimento e desempenho de aplicações cotidianas e de programas que requerem alto desempenho. A proficiência em algoritmos, as suas complexidades inerentes e métodos de refinamento são, portanto, cruciais para qualquer desenvolvedor.

Com a SDK, ferramenta disponibilizada pelo VS Code, foi desenvolvido um analisador estático de código. Este analisador converte código em C para uma AST, permitindo a análise e cálculo da complexidade de cada componente do código. Contudo, há limitações inerentes a tal abordagem. Quando essas limitações forem encontradas, foi disponibilizada a alternativa de análise por IA Generativa LLM, que se mostrou muito mais precisa.

É possível incrementar a ferramenta perante a AST, para tornar a análise cada vez mais completa e dessa maneira evitando o custo da IA. Além disso, a própria IA poderia ter seu modelo aprimorado para ficar especializada apenas nessa funcionalidade, de estimar a complexidade. Também seria possível usar outros métodos de inteligência artificial, que demandariam menos recursos computacionais, como *Machine Learning* ou uma análise simbólica. Além disso, hoje a extensão permite ao usuário selecionar o método de análise. Seria mais intuitivo ter uma forma híbrida que detecte o melhor método para cada análise e qual pode providenciar o resultado mais adequado, sem a necessidade de interpretação do usuário.

REFERÊNCIAS

ALBERT, E. *et al.* Costa: Design and implementation of a cost and termination analyzer for java bytecode. In: BOER, F. S. de *et al.* (Ed.). **Formal Methods for Components and Objects**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 113–132. ISBN 978-3-540-92188-2.

ANDERSON, W. **Extensions using CodeLens**. Microsoft, 2017. Disponível em: <https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup>. Acesso em: 27 fev. 2024.

BARBOSA, M. A. d. C. **ANAC - Uma Ferramenta para a Automatização da Análise da Complexidade de Algoritmos**. Dissertação (Mestrado) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2001.

CLANG. **Introduction to the Clang AST**. 2007. Disponível em: <https://clang.llvm.org/docs/IntroductionToTheClangAST.html>. Acesso em: 21 jun. 2024.

CODESCENE. **CodeScene**. 2024. Disponível em: <https://github.com/codescene-oss/codescene-vscode>. Acesso em: 09 abr. 2024.

CORMEN, T. H. *et al.* **Introduction to Algorithms**. 4. ed. [S.l.]: The MIT Press, 2022. ISBN 9780262046305.

COSTA, E. J. F. *et al.* Um avaliador automático de eficiência de algoritmos para ambientes educacionais de ensino de programação. In: **Anais do Computer on the Beach**. Florianópolis, SC: Universidade do Vale do Itajaí, 2014. ISBN 978-85-7696-120-8.

DIVERIO, T. A.; MENEZES, P. B. **Teoria da Computação**: Máquinas universais e computabilidade. 3. ed. [S.l.]: Bookman, 2011. v. 5. (Série livros didáticos informática ufrgs, v. 5). ISBN 9788577808243.

EMBOLD. **Embold | Static Code Analysis Platform**. 2024. Disponível em: <https://embold.io/>. Acesso em: 06 abr. 2024.

EXTENSãO Big O. 2024. Disponível em: <https://github.com/francopan/code-performance-analyzer-ext>. Acesso em: 18 nov. 2024.

GITKRAKEN. **GitLens**. 2024. Disponível em: <https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens>. Acesso em: 12 abr. 2024.

HILBERT, D. **Mathematische Probleme**. [s.n.], 1900. Disponível em: <https://www.digizeitschriften.de/id/252457811_1900|log37>. Acesso em: 01 jul. 2024.

HILBERT, D.; ACKERMANN, W. **Grundzüge der theoretischen Logik**. 1. ed. [S.l.]: Springer, 1928.

INTEL. **Intel Breakthroughs Propel Moore's Law Beyond 2025**. 2024. Disponível em: <https://www.intel.com/content/www/us/en/newsroom/resources/moores-law.html>. Acesso em: 22 abr. 2024.

JETBRAINS. **Inspectopedia Documentation**. 2024. Disponível em: <https://www.jetbrains.com/help/inspectopedia/WhatIsNewInspections.html>. Acesso em: 06 abr. 2024.

JETBRAINS. **Qodana: Static Code Analysis Tool by JetBrains**. 2024. Disponível em: <https://www.jetbrains.com/qodana/>. Acesso em: 06 abr. 2024.

KISS, T. **CodeMetrics**. 2024. Disponível em: <https://github.com/kisstkondoros/codemetrics>. Acesso em: 09 abr. 2024.

KNUTH, D. E. **The Art of Computer Programming**: Fundamental algorithms. 3. ed. [S.l.]: Addison-Wesley Professional, 1997. v. 1. ISBN 9780201896831.

LAUBE, U.; NEBEL, M. E. Maximum likelihood analysis of algorithms and data structures. **Theoretical Computer Science**, v. 411, n. 1, p. 188–212, 2010. ISSN 0304-3975. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0304397509006732>. Acesso em: 07 abr. 2024.

MCCABE, T. J. A complexity measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 308–320, 1976.

MICROSOFT. **Language Server Protocol**. 2024. Disponível em: <https://microsoft.github.io/language-server-protocol/>. Acesso em: 09 abr. 2024.

MICROSOFT. **Microsoft Copilot | IA da Microsoft**. 2024. Disponível em: <https://www.microsoft.com/pt-br/microsoft-copilot>. Acesso em: 10 jun. 2024.

MICROSOFT. **Publishing Extensions**. Microsoft, 2024. Disponível em: <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>. Acesso em: 06 dez. 2024.

MICROSOFT. **Your First Extension | Visual Studio Code Extension API**. Microsoft, 2024. Disponível em: <https://code.visualstudio.com/api/get-started/your-first-extension>. Acesso em: 27 fev. 2024.

NAVEED, H. *et al.* **A Comprehensive Overview of Large Language Models**. 2024. Disponível em: <https://arxiv.org/abs/2307.06435>. Acesso em: 21 ago. 2024.

OLLAMA. **Ollama**. 2024. Disponível em: <https://ollama.com/>. Acesso em: 01 jul. 2024.

PETZOLD, C. **The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine**. 1. ed. [S.l.]: John Wiley and Sons, 2008. ISBN 9780470229057.

SHERINE, A. *et al.* **Algorithm and Design Complexity**. 1. ed. [S.l.]: CRC Press, 2023. ISBN 1032409320.

SONAR. **Code Quality, Security and Static Analysis Tool with SonarQube | Sonar**. 2024. Disponível em: <https://www.sonarsource.com/products/sonarqube/>. Acesso em: 06 abr. 2024.

SONAR. **Quality gates**. 2024. Disponível em: <https://docs.sonarsource.com/sonarcloud/improving/quality-gates/>. Acesso em: 27 mai. 2024.

TARJAN, R. E. Amortized computational complexity. **SIAM Journal on Algebraic and Discrete Methods**, Society for Industrial and Applied Mathematics, v. 6, n. 2, p. 307–317, 1985.

TEAM, G. *et al.* **Gemini: A Family of Highly Capable Multimodal Models**. 2024. Disponível em: <https://arxiv.org/abs/2312.11805>. Acesso em: 10 jun. 2024.

TOSCANI, L. V.; VELOSO, P. A. S. Uma metodologia para cálculo da complexidade de algoritmos. In: **Brazilian Symposium on Software Engineering**. [s.n.], 1990. Disponível em: <https://api.semanticscholar.org/CorpusID:266874156>.

TOSCANI, L. V.; VELOSO, P. A. S. **Complexidade de algoritmos**. 3. ed. [S.l.]: Bookman, 2012. v. 13. (Série livros didáticos informática ufrgs, v. 13). ISBN 9788540701380.

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. **Proceedings of the London Mathematical Society**, Association for Symbolic Logic, v. 42, n. 1, p. 230–265, 1936.

VAZ, R. *et al.* Automated big-o analysis of algorithms. In: **2017 International Conference on Nascent Technologies in Engineering (ICNTE)**. [S.l.: s.n.], 2017. p. 1–6.

APÊNDICE A – RESULTADOS

Função	**AST**	**AST OK**	**LLM**	**LLM OK**	**Esperado**
constant_loop	1	V	n	F	1
arithmetic_operations	1	V	1	V	1
constant_conditional	1	V	1	V	1
fixed_array_operations	1	V	1	V	1
constant_switch	1	V	1	V	1
linear_loop	n	V	n	V	n
linear_with_condition	2^n	F	n	V	n
linear_cumulative_sum	1	F	n	V	n
linear_independent_loops	2^n	F	n	V	n
quadratic_loop	n^2	V	n^2	V	n^2
quadratic_array_operations	$1, 2^n$	F	n^2	V	n^2
quadratic_with_condition	2^n	F	n^2	V	n^2
quadratic_with_arithmetic	2^n	F	n^2	V	n^2
quadratic_independent_loops	2^n	F	n^2	V	n^2
cubic_loop	n^3	V	n^3	V	n^3
cubic_array_operations	$1, 2^n$	F	n^3	V	n^3
cubic_with_condition	2^n	F	n^3	V	n^3
cubic_with_arithmetic	2^n	F	n^3	V	n^3
cubic_independent_operations	2^n	F	n^3	V	n^3
exponential_loop	2^n	V	2^n	V	2^n
exponential_subsets	2^n	V	2^n	V	2^n
exponential_recursive	1	F	2^n	V	2^n
exponential_backtracking	1	F	2^n	V	2^n
factorial	1	F	n	V	n
factorial_iterative	n	V	n	V	n
factorial_memo	1	F	n	V	n
factorial_tail_recursive	1	F	n	V	n
factorial_dynamic	2^n	F	n	V	n
logarithmic_loop	n	F	$\log(n)$	V	$\log(n)$
logarithmic_recursive	1	F	$\log(n)$	V	$\log(n)$
binary_search	1	F	$\log(n)$	V	$\log(n)$
logarithmic_nested_loops	n	F	$O(\log(n) \cdot \log(n))$	V	$O(\log(n) \cdot \log(n))$
Acurácia	-	**34,37%**	-	**96,87%**	-

Tabela 1 – Resultados dos Testes de Complexidade Big O